MASTER EN SCIENCES INFORMATIQUES

Conception de workflow par des langages de coordination

Cordier, Stéphanie

Award date:
2008

Awarding institution:
Universite de Namur

Facultés Universitaires Notre-Dame de la Paix, Namur
Institut d'informatique
Année académique 2006-2007

Conception de workflow par des langages de coordination

Stéphanie Cordier

Mémoire présenté en vue de l'obtention du grade de licenciée en Informatique

Résumé

Dans ce mémoire, nous allons nous intéresser aux workflow et à leur modélisation à l'aide de langages de coordination. Le workflow est une problématique actuelle et récurrente pour la plupart des organisations. Celles-ci sont confrontées au problème de gérer efficacement les informations qu'elles détiennent soit explicitement dans des bases de données soit implicitement, par exemple au travers de courriers électroniques. En outre, elles aimeraient disposer d'outils pour gérer le flux de ces informations. Dans ce mémoire, nous étudions l'expressivité des langages de coordination au travers de la conception de tels systèmes.

mots-clés : workflow, langage de coordination, YAWL, REO

Abstract

In this thesis we will look at workflows and their models using coordination languages. Workflows are a current subject of interest and are recurrent in most enterprises. They face the problem to handle efficiently the mass of information that they compile explicitly (e.g. in databases) or implicitly (e.g. within e-mails). Moreover, they wish to have tools to handle this flow of information. In this thesis we study the expressiveness of coordination languages through the conception of such systems.

Keywords : workflow, coordination language, YAWL, REO

Ce mémoire n'est pas le fruit d'un travail solitaire. La longueur des remerciements qui suivent n'est pourtant pas assez représentative de la gratitude que je voudrais témoigner à chacune des personnes qui ont contribué à l'accomplissement de ce mémoire.

Mes remerciements s'adressent en premier lieu à mon promoteur, M. Jean-Marie Jacquet, pour l'aide qu'il m'a apportée tout au long de ce mémoire. Je le remercie surtout pour sa disponibilité et ses nombreuses remarques toujours pertinentes.

Je remercie de tout mon cœur mon conjoint pour son soutien quotidien. Ce mémoire n'aurait certainement jamais vu le jour sans son appui.

Je remercie également mes amis pour la patience dont ils ont fait preuve cette année. Je tiens aussi à remercier Slimane, Dina, Nicolas et Evelyne pour leurs relectures attentives.

Enfin, ces remerciements ne pourraient être complets sans y associer ma famille. Merci de toujours avoir été là pour moi.

Stéphanie

Table des matières

Introduction

Les workflow sont depuis plusieurs années de plus en plus présents dans tout type d'applications. On les retrouve dans des systèmes aussi variés que la demande de permis de bâtir, la prise en charge d'un appel dans un centre d'appels, la facturation d'un patient dans un hôpital, l'achat d'un produit sur un site d'e-commerce, l'acceptation d'un article scientifique dans une revue académique, etc.

Bien que les workflow soient présents sous différentes formes et dans des secteurs forts différents, ils ont beaucoup d'éléments en commun. Ainsi, on retrouvera dans tous ces workflow, des éléments tels que des tâches, des flux, des données qui doivent transiter, des ressources qui seront affectées, etc. C'est pourquoi, il est possible d'implémenter des moteurs de workflow en restant générique.

Un autre point commun de ces différents exemples de workflow est le fait qu'ils sont le plus souvent constitués de plusieurs tâches. Ces tâches peuvent (et/ou doivent) s'exécuter en parallèle. On retrouve alors un aspect de coordination essentiel.

Le sujet de ce mémoire touche directement aux points abordés dans les deux paragraphes précédents. D'une part, on analyse dans ce mémoire l'implémentation d'un moteur de workflow. D'autre part, on utilise les langages de coordination pour nous aider dans cette implémentation.

L'un des objectifs de ce travail est donc de déterminer l'intérêt des langages de coordination dans l'implémentation d'un tel moteur. Il nous faudra donc discerner les avantages et les inconvénients de notre choix d'implémentation.

La première étape de notre mémoire a été de nous familiariser avec les moteurs de workflow. Dans le monde des workflow, il existe de nombreuses normes et différents standards. Nous présentons donc l'état de l'art dans le premier chapitre. Une autre étape importante a été de réaliser le même travail pour les langages de coordination. Nous avons aussi dû choisir parmi les langages existants ceux qui étaient les plus appropriés dans notre cas.

Pour analyser et implémenter un moteur de workflow, il nous faut choisir une approche et une méthodologie. L'implémentation d'un tel logiciel n'est pas une mince affaire et il est donc crucial d'avoir une approche structurée. Dans le premier chapitre, on verra qu'il est possible d'analyser un workflow sous quatre perspectives différentes qui sont le contrôle de flux, les données, les ressources et l'opérationnelle [van der Aalst et Hofstede, 2002].

Nous avons opté pour une implémentation itérative. Chaque itération se base sur une des quatre perspectives. La méthodologie utilisée est une variante de RUP (Rational Unified Process) et le langage de modélisation choisi est l'UML (Unified Modeling Language). Nous avons choisi le langage Java comme interface de notre implémentation. À celui-ci, nous avons greffé le langage de coordination Reo [Kan, 2005] pour la gestion du flux au sein du moteur de workflow. Finalement, notre implémentation s'est aussi fortement inspirée du moteur de workflow YAWL (Yet Another Workflow Language) [van der Aalst et Hofstede, 2002]. On utilise le langage de modélisation

de YAWL pour modéliser le workflow. Ce choix s'appuie sur une étude de plusieurs autres langages [van der Aalst *et al.*, 2003].

L'apport de ce mémoire est donc double. Premièrement, nous présentons au travers d'une analyse complète les différentes perspectives qui caractérisent un moteur de workflow. Deuxièment, nous utilisons les langages de coordination et YAWL dans l'implémentation du moteur.

Chapitre 1

Workflow

Ce chapitre présente une vue globale de plusieurs concepts récurrents dans le domaine du workflow. Nous commençons par une définition formelle de workflow qui est illustrée par un exemple. Par la suite, nous expliquons les différents types de workflow. Un workflow peut aussi être abordé sous différentes perspectives. Nous présentons donc les quatre perspectives communément acceptées dans la littérature. Ensuite, nous présentons un aperçu des différentes organisations qui traitent des workflow et des différents standards qui ont été définis. Enfin, nous terminons ce chapitre par un aperçu d'un moteur de workflow qui se trouve sur le marché.

1.1 Définition

On appelle « *workflow*[1] » la modélisation d'un flux d'informations au sein d'une organisation afin d'accomplir un processus métier, aussi appelé processus opérationnel ou processus de l'entreprise. Il peut être partiel ou complet. Un exemple de workflow est le traitement d'un accident automobile par une compagnie d'assurance.

Il est important de noter la différence entre un workflow et une instance de workflow. Le workflow représente la définition du processus métier au sein du moteur de workflow alors qu'une instance de workflow représente un cas de ce workflow.

Le *moteur de workflow*[2] est « un dispositif logiciel permettant d'exécuter des instances de workflow ». Par abus de langage, on peut appeler ce dispositif logiciel tout simplement « *workflow* ». Celui-ci pourrait être traduit en français par « gestion électronique des processus métiers ».

Le moteur de workflow connaît les tâches, la procédure appliquée pour chaque cas qu'il sait traiter et les règles de gestion en vigueur. En pratique, il va faciliter le suivi et la coordination entre les acteurs en leur affectant une tâche en fonction de leurs qualifications ainsi que des délais et des modes de validation. Il va également leur donner les informations et les outils nécessaires pour la réaliser au mieux.

Enfin, la WfMC a introduit une nuance entre un moteur de workflow et un système de gestion de workflow. Un *système de gestion de workflow*[2] est « le dispositif logiciel pouvant être constitué d'un ou plusieurs moteurs de workflow et dont le but est de créer, de gérer et d'exécuter des instances de workflow. En plus de gérer l'exécution de tels processus, il permet

[1] Site wikipedia, `http://fr.wikipedia.org/wiki/Workflow` (visité le 26/06/2007).

[2] Site wikipedia, `http://fr.wikipedia.org/wiki/Moteur_de_workflow` (visité le 26/06/2007).

de s'interfacer avec des outils d'administration, des applications clientes, d'autres systèmes de gestion de workflow, etc. »

1.2 Exemple

Pour illustrer notre définition, nous proposons l'exemple d'un workflow assez simple. Celui-ci se compose de cinq tâches et décrit la préparation d'un voyage (d'affaire ou touristique) dans une agence de voyages.

Les cinq tâches sont :

Enregistrement des données : on encode les coordonnées des différents passagers ainsi que les données relatives au voyage. Le personnel de l'agence demande entre autres le nom, l'adresse, la nationalité, la destination, les dates de départ et de retour, etc.

Réservation d'un avion : on réserve le ticket d'avion avec les éventuelles correspondances.

Réservation d'un d'hôtel : on réserve une ou plusieurs chambres d'hôtel.

Réservation d'une voiture : on réserve une voiture de location pour la période souhaitée.

Paiement : on paie le montant total à l'agence de voyages.

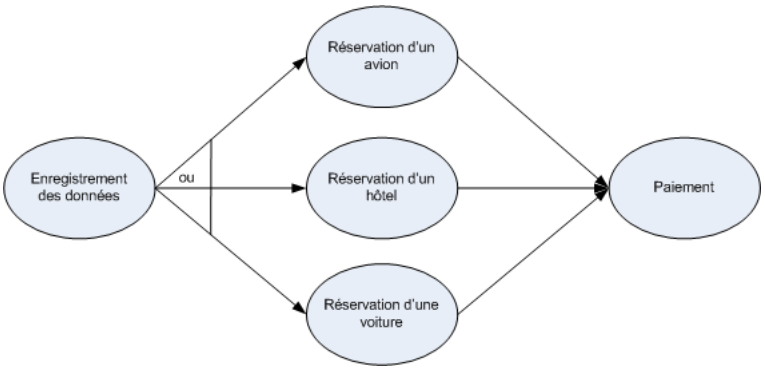

Figure 1.1 – Exemple de workflow pour une agence de voyages

La figure 1.1 présente le workflow au complet. Comme on le remarque sur cette figure, un client n'est pas obligé d'effectuer les trois réservations. S'il le désire, il peut uniquement réserver une voiture ou un ticket d'avion. Ces trois tâches sont donc reliées entre elles par un « *OU* ». C'est-à-dire qu'on peut effectuer soit une, soit deux, soit trois réservations. Néanmoins, il faut au moins une réservation pour passer à l'étape suivante dans le workflow.

Notons aussi que nous avons délibérément simplifié cet exemple afin de présenter l'idée générale. Nous ne prenons donc pas en compte de multiples escales lors du voyage ou le paiement éventuel d'un acompte.

Le workflow ainsi défini peut être à présent instancié pour chaque nouveau client. Ainsi, pour deux clients différents, nous aurons deux instances différentes.

1.3 Types de workflow

On distingue classiquement quatres familles de workflow.
- les workflow de production ;
- les workflow administratif ;
- les workflow ad-hoc ;
- les workflow coopératif.

La distinction entre ces quatre familles se fait sur base de deux critères [Levan, 1999]. Le premier critère détermine si le workflow est centré sur le document ou sur un groupe d'utilisateurs. Le deuxième critère détermine si le flux dans le workflow est contrôlé par un moteur de workflow ou si l'information transite via une messagerie.

La figure 1.2 illustre les deux critères qui distingue ces quatre familles.

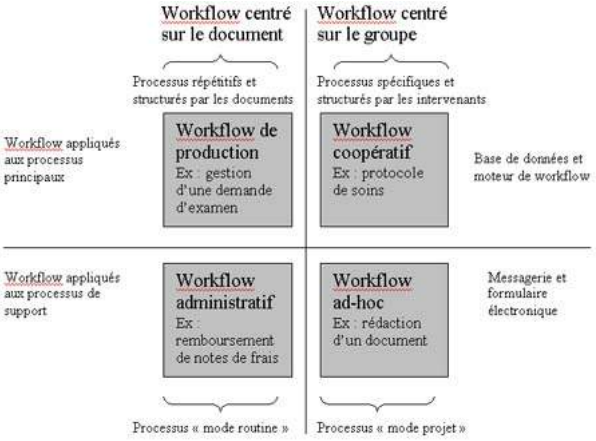

Figure 1.2 – Typologie fonctionnelle des applications de workflow [Levan, 1999]

1.3.1 Workflow de production

Le workflow de production correspond à des processus métiers connus de l'entreprise et faisant l'objet de procédures pré-établies. Ceci implique donc peu de changements dans le temps et des transactions répétitives. C'est pourquoi, le workflow est contrôlé par un moteur et que l'échange se fait grâce à des documents. Il recherche la performance et la rigueur d'exécution dans un contexte de flux importants et de temps de réponses proches de ceux du traitement transactionnel. Le cheminement du workflow est plus ou moins figé. On peut citer comme exemple la gestion des sinistres dans une compagnie d'assurance.

1.3.2 Workflow administratif

Le workflow administratif est identique au précédent sur le fait qu'il est basé sur des documents et donc sur des processus « *mode-routine* ». Celui-ci est dédicacé pour un travail administratif répétitif et est beaucoup plus flexible afin de faciliter la définition du processus. Il est basé en général sur une infrastructure de messages avec un routage de formulaires. Ce workflow est également plus transversal dans l'entreprise dans le sens où il se retrouve à tous les niveaux de celle-ci. On peut citer comme exemple l'approbation d'une note de frais.

1.3.3 Workflow ad-hoc

Le workflow ad-hoc est basé sur un modèle collaboratif dans lequel les acteurs interviennent dans la décision du cheminement pour une action particulière à un instant T. Le cheminement du workflow est donc dynamique. Il est également basé sur un système de messagerie. On peut citer comme exemple l'offre de postes en interne dans une entreprise.

1.3.4 Workflow coopératif

Le workflow coopératif établit, définit et gère les éléments de procédures évolutifs liés à un groupe de travail restreint dans l'entreprise. Il est très souple et spécifique. Il recherche avant tout la facilité d'emploi et la souplesse des procédures que l'utilisateur doit pouvoir redéfinir aisément à tout instant au sein du moteur de workflow. Chaque système fait appel à des techniques différentes. On peut citer comme exemple la rédaction de protocoles opératoires.

1.4 Perspectives

Dans le cadre de ce mémoire, nous voulons montrer qu'il est possible via des langages de coordination d'implémenter un moteur de workflow. Afin de faciliter notre travail, nous reprenons l'idée de [van der Aalst et Hofstede, 2002] qui consiste à examiner les workflow sous 4 perspectives différentes :

Perspective de contrôle de flux décrit les tâches et l'ordre d'exécution. Une tâche élémentaire est une unité atomique du travail. Une tâche composée est l'ordre d'exécution d'un ensemble de tâches.

Perspective de données traite des données comme des documents par exemple. On les qualifie comme les pré et les post conditions d'une tâche. Elle se trouve au-dessus de la perspective de contrôle de flux.

Perspective de ressources fournit la structure de l'organisation dans lequel le moteur de workflow va être utilisé. C'est dans cette perspective qu'on va définir les membres du personnel et leurs rôles dans l'accomplissement d'une ou de plusieurs tâches du workflow.

Perspective opérationnelle décrit les actions élémentaires à exécuter par une tâche. Cette perspective permet d'établir un lien avec des applications externes.

1.5 Organisation de standardisation

Il existe deux organisations de standardisation, la WfMC et la WARIA.

1.5.1 WfMC - Workflow Management Coalition

La Workflow Management Coalition[3] est une organisation internationale à but non lucratif, fondée en août 1993. Elle regroupe plus de 300 membres. Parmi ceux-ci, on peut citer IBM, Sun Microsystems, Oracle et SAP. La coallition est rapidement devenue le premier organisme de standardisation dans le domaine du workflow. Ces membres réunissent des vendeurs de solutions de workflow, des utilisateurs, des analystes et des groupes de recherche.

[3]Site du Workflow Management Coallition : `http://www.wfmc.org` (visité le 26/06/2007).

Sa mission est la promotion et le développement de l'utilisation du workflow à travers des normes acceptées par tous. Celles-ci recouvrent la terminologie, l'interopérabilité et la connectivité entre les différents produits.

Modèle de référence du workflow de WfMC

La figure 1.3 montre le modèle de référence du workflow de la WfMC [Hollingsworth, 1995].

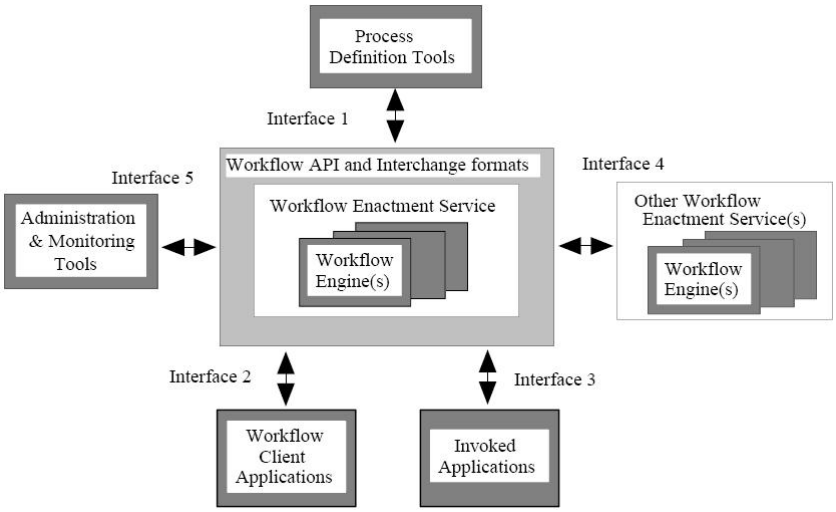

Figure 1.3 – Modèle de référence du workflow [Hollingsworth, 1995]

Celui-ci est basé sur cinq interfaces :

Process Definition Tools Interface (1) est la définition d'une interface standard pour la définition de processus et les outils de modélisation.

Workflow Client Application Interface (2) est la définition d'une API pour les applications clientes afin de pouvoir demander des services du moteur de workflow et de contrôler la progression de ces processus et de ces activités.

Invoked Application Interface (3) est la définition d'interfaces standards pour permettre au moteur de workflow d'appeler une variété d'applications, à travers un *software* d'agent commun.

Workflow Interoperability Interface (4) est la définition des modèles d'interopérabilité et des standards correspondants afin d'aider à la collaboration.

Administration and Monitoring Tools Interface (5) est la définition du monitoring et des fonctions de contrôle.

Standards de WfMC

La WfMC a développé deux standards, XPDL et Wf-XML.

XPDL (XML Process Definition Language)[4] permet « de définir des processus métiers qui seront ensuite utilisés par le moteur de workflow ». Il correspond à l'interface 1 de la figure 1.3.

[4]Site wikipedia, `http://fr.wikipedia.org/wiki/XML_Process_Definition_Language` (visité le 26/06/2007).

Wf-XML[5] permet de fournir une interface d'interopérabilité pour que les systèmes de workflow puissent communiquer entre eux. « Il étend les services SOAP (Simple Object Access Protocol) pour inclure des fonctionnalités asynchrones de gestion d'un workflow »[6]. Il correspond à l'interface 4 de la figure 1.3.

1.5.2 WARIA - Workflow And Reengineering International Association

La Workflow and Reengineering International Association[7] est la collaboration entre l'OMG (Object Management Group) et la BPMI (Business Process Management Initiative).

La WARIA identifie et étudie les problèmes rencontrés par les utilisateurs de workflow et par ceux qui entendent restructurer leur organisation.

L'objectif de la WARIA est de clarifier les liens entre les problématiques de *reengineering*, de gestion des processus, de workflow et de commerce électronique. Et ce, par le partage d'expériences, d'évaluations de produits, par la mise en relation des fournisseurs avec les utilisateurs et par la formation.

1.6 YAWL - Yet Another Workflow Language

1.6.1 Introduction

Avant de commencer à implémenter notre moteur de workflow, nous avons voulu en étudier un. Nous avons choisi YAWL[8] qui est l'acronyme de Yet Annother Workflow Language. C'est un effort commun de l'université d'Eindhoven (Pays-Bas) et Queensland University of Technology (Australie). Les auteurs ont développé ce langage après avoir constaté les limitations des autres langages de workflow. L'une de ces limitations était qu'il était possible d'interpréter un même workflow de multiples façons différentes. Dès lors, le résultat du workflow pouvait différer selon le langage utilisé. Les auteurs proposent donc une sémantique rigoureuse afin de pallier à ce défaut.

Notons que leur langage est open-source alors que la plupart des autres langages sont propriétaires et donc ils sont souvent liés à un outil bien spécifique. YAWL n'a pas ce défaut et est indépendant de l'outil sous-jacent utilisé.

YAWL s'est inspiré des réseaux de Pétri mais malheureusement s'est heurté à quelques problèmes. C'est pourquoi les auteurs n'ont pas simplement étendu les réseaux de Petri. Sa sémantique a été définie en terme de systèmes à transitions et a été enrichie avec des mécanismes additionnels pour tenir compte d'un appui plus direct et intuitif aux patterns de workflow [van der Aalst et Hofstede, 2002].

[5] Site technweb.com, `http://www.techweb.com/encyclopedia/defineterm.jhtml?term=Wf-XML` (visité le 26/06/2007).

[6] Site techno-science.net, `http://www.techno-science.net/?onglet=glossaire&definition=1330` (visité le 26/06/2007).

[7] Site du Workflow And Reengineering International Association : `http://www.waria.com/index-waria.html` (visité le 26/06/2007).

[8] Site de YAWL : `http://yawlfoundation.org/` (visité le 26/06/2007).

1.6.2 Architecture

La figure 1.4 représente l'architecture de YAWL [Persson et Stirna, 2004].

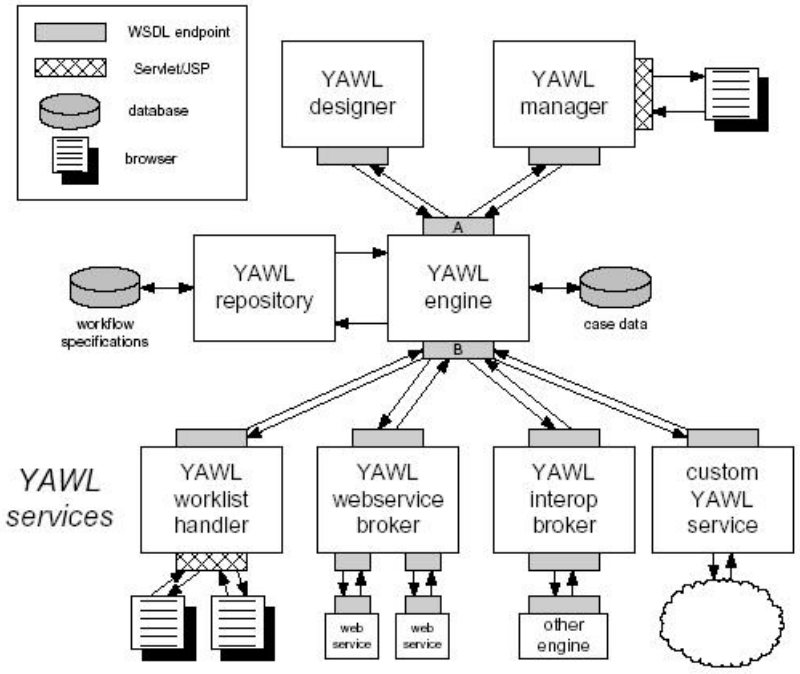

Figure 1.4 – Architecture de YAWL [Persson et Stirna, 2004]

Le noyau du système est le moteur de workflow de YAWL (« *YAWL engine* »). Il peut instancier des spécifications de workflow qui sont dessinées avec le « *YAWL designer* ». Ces spécifications de workflow sont gérées par le « *YAWL repository* », auquel on peut acceder via le moteur de workflow.

L'environnement du système YAWL est composé de « *YAWL services* ». Les services YAWL sont inspirés du paradigme de *web service*. Ceux-ci concernent aussi bien les utilisateurs que les applications.

Il y a quatre services YAWL :

1. « *YAWL worklist handler* » sert à assigner du travail aux utilisateurs du système. Pour chaque utilisateur, le travail est présenté sous forme de liste. Ce dernier peut l'accepter et avertir le système une fois que le travail est réalisé.
2. « *YAWL web service broker* » agit comme médiateur entre le moteur de workflow et les web services externes qui peuvent être invoqués par le moteur pour déléguer une tâche.
3. « *YAWL interoperability broker* » est un service conçu pour interconnecter différents moteurs de workflow.
4. « *Custom YAWL services* » est un service client qui connecte le moteur avec une entité de l'environnement du système comme par exemple une imprimante.

Notons qu'il est également possible qu'il y ait des services multiples du même type, par exemple une work-list par zone géographique.

Les spécifications de workflow sont gérées par le « *YAWL repository* » et les instances de workflow sont gérées par le « *YAWL engine* ». Enfin, YAWL comporte un dernier composant :

« *YAWL manager* ». C'est un outil de gestion pour manipuler les instances de workflow, fournir l'état d'une exécution d'une instance du workflow, et détailler des données sur des instances complètes.

« *YAWL engine* » a deux interfaces. L'interface A capture les interactions entre « *YAWL designer* » et « *YAWL manager* » d'une part, et « *YAWL engine* » de l'autre. Elle correspond à « *Process Definition Tools Inteface (1)* » et « *Administration and Monitoring Tools Interface (5)* » du modèle de référence de WfMC. L'interface B capture les interactions entre « *YAWL services* » et « *YAWL engine* ». Elles correspondent à « *Workflow Client Application Interface (2)* », « *Invoked Application Interface (3)* » et « *Workflow Interoperability Interface (4)* » du modèle de référence de WfMC.

Les deux interfaces A et B sont spécifiées en WSDL (Web Service Description Language). Les utilisateurs ont accès au système YAWL via un navigateur Web.

Le moteur YAWL traite des flux de contrôle et des données mais de manière implicite pour l'utilisateur. Les système de gestion de workflow traditionnel demande « Quoi », « Quand », « Comment » et « Par qui ». Le système YAWL demande seulement « Quoi » et « Quand » et c'est le service de YAWL qui s'occupe du « Comment » et « Par qui ». Cette séparation permet d'avoir un moteur de workflow très efficace en tenant compte des fonctionnalités des utilisateurs.

1.7 Conclusion

Nous venons d'établir un état de l'art de la partie workflow du mémoire. Ce chapitre est un rassemblement des concepts nécessaires pour poursuivre le travail. Nous avons tout d'abord défini ce qu'est un workflow, puis nous avons enchaîné avec les différents types de workflow [Levan, 1999] et les différentes perpectives [van der Aalst et Hofstede, 2002]. Nous avons également présenté deux organismes de standardisation, la première est la WfMC et la seconde est WARIA. Enfin, nous avons terminé ce chapitre en étudiant le moteur de workflow de YAWL [van der Aalst et Hofstede, 2002].

Dans le chapitre suivant nous présentons un état de l'art des langages de coordination.

Chapitre 2

Langages de coordination

Ce chapitre présente une vue globale des langages de coordination. Nous commençons par une définition formelle d'un modèle et d'un langage de coordination. Ensuite, nous présentons les deux catégories de langage de coordination : les langages orientés événements et ceux orientés données. Enfin, nous terminons par une étude d'un langage de coordination pour chacune de ces catégories afin d'en saisir le mécanisme.

2.1 Définition

La *coordination* est un concept compris par la majorité des gens. On sait reconnaître une situation où il y a une bonne coordination, mais on sait surtout reconnaître l'absence de coordination. Par exemple, on se rend compte du manque de coordination lorsqu'on doit en subir les conséquences, comme une collision ou un retard.

Nous retrouvons dans la littérature plusieurs définitions pour la *coordination* :
- « Le processus de construction de programmes en assemblant ensemble des pièces actives » [Carriero et Gelernter, 1989] ;
- « L'intégration et l'ajustement harmonieux des différents efforts de travaux individuels à travers l'accomplissement d'un but plus large » [Singh, 1992] ;
- « L'acte des dépendances de gestion entre les activités » [Malone et Crowston, 1994].

Un *modèle de coordination* fournit un « *framework* » dans lequel l'interaction des agents individuels peut être exprimée [Wegner, 1997]. Ceci couvre les aspects de
- la création et de la destruction d'agents ;
- la communication parmi les agents ;
- la distribution spatiale des agents ;
- la synchronisation et la distribution des actions dans le temps.

Un modèle de coordination est généralement caractérisé par un triplet (E,M,L) :

E sont les entités à coordonner : les agents actifs qui sont coordonnés et les modules d'une architecture de coordination (agents, processus, tuples, atomes, etc) ;

M est le médium de coordination : les médias permettant la coordination des entités inter-agent. Ils servent à agréger un ensemble d'agents pour former une configuration (canaux, variables partagées, espaces de tuples, etc) ;

L est le type de coordination : les règles d'action pour coordonner les entitités (accès, contraintes de synchronisation, etc).

Un langage de coordination est « l'incarnation linguistique d'un modèle de coordination » [Carriero et Gelernter, 1989]. Il implémente un modèle de coordination. Il doit combiner orthogonalement deux modèles : un pour la coordination (les actions inter-agents) et un pour le calcul séquentiel (les actions intra-agents).

On peut diviser les modèles de coordination en deux grandes familles [Arbab et Talcott, 2002] : les orientés données (« *data-driven* ») et les orientés événements (« *control-driven* »).

Les modèles orientés données (« *data-driven* ») ont pour principe commun l'utilisation d'une mémoire partagée pour la communication des données.

Les modèles orientés événements (« *control-driven* ») utilisent les canaux et les ports pour leurs communications.

En conclusion, le modèle de coordination fournit la sémantique, alors qu'un langage de coordination fournit une syntaxe pour utiliser le modèle de coordination.

Nous présentons dans la suite de ce chapitre, un language de coordination par type de modèles. Nous avons choisi Reo pour la coordination par événement et KLAVA pour la coordination par espace de tuple.

2.2 Reo : coordination par événement

Reo [Kan, 2005], de Farhad Arbab du CWI[1] (Centrum voor Wiskunde en Informatica), est un modèle de coordination orienté événements et basé sur la notion de canaux (« *channel* ») qui relient et coordonnent différentes activités par des connecteurs (« *connector* »).

Nous avons choisi d'utiliser Reo pour plusieurs raisons. Tout d'abord, nous avons trouvé une distribution récente du langage (2004) ce qui n'était pas le cas des autres langages que nous avons examinés (Manifold, STL)[2]. De plus, Reo avait l'avantage d'être écrit en Java et de bénéficier d'une distribution *open source*.

Nous avons utilisé la distribution Reolite de Dave Clarke[3].

2.2.1 Concepts

Le canal (« *channel* ») est le plus simple des connecteurs (« *connector* ») en Reo. Il possède deux extrémités qui se nomment « *source end* » et « *sink end* ». La première est l'entrée et la deuxième est la sortie du canal. Nous prendrons comme convention *source* et *destination* pour la suite du chapitre.

Un connecteur est un ensemble de canaux organisés en graphe avec un comportement bien défini pour effectuer des opérations d'entrée et de sortie. Il permet la communication entre plusieurs instances d'un composant. Une extrémité d'un canal peut être attachée à seulement une instance d'un composant à la fois.

[1] Site du Centrum voor Wiskunde en Informatica d'Amsterdam aux Pays-Bas : `http://www.cwi.nl/` (visité le 27/06/2007).

[2] Par exemple, Manifold du CWI d'Amsterdam aux Pays-Bas date de 1993.

[3] Disponible sur le site : `http://homepages.cwi.nl/~dave/reolite/` (visité le 06/07/2007).

Une instance d'un composant permet d'échanger de l'information avec son environnement. Elle peut utiliser un ou plusieurs ports d'entrées et de sorties. Ce port permet à l'instance du composant de se connecter au connecteur Reo. La figure 2.1 représente une instance d'un composant avec 4 ports.

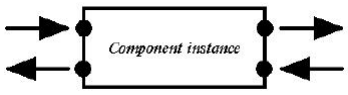

Figure 2.1 – Instance d'un composant avec 4 ports [Kan, 2005]

2.2.2 Méthode d'un composant

Voici la liste des méthodes possibles d'un composant :

create Créer un nouveau canal et renvoyer les identifiants de deux extrémités.

forget Changer la valeur d'une extrémité du canal pour qu'elle ne référence plus l'extrémité du canal.

move Déplacer l'extrémité d'un canal à un nouvel endroit.

connect Connecter l'extrémité d'un canal à une instance d'un composant qui traite cette opération.

disconnect Déconnecter l'extrémité d'un canal à une instance d'un composant qui traite cette opération.

wait Attendre que le booléen de la condition devient *true*.

join Joindre deux extrémités de canaux.

split Dédoubler le nœud attaché à l'extrémité d'un canal en deux nouveaux nœuds.

hide Cacher le nœud attaché à l'extrémité du canal de sorte qu'il ne puisse pas être modifiée par une autre opération.

read Copier la donnée à partir de l'extrémité du canal sans l'enlever.

take Copier la donnée à partir de l'extrémité du canal et ensuite l'enlever.

write Écrire une donnée sur l'extrémité du canal si elle peut l'accepter.

2.2.3 Channels

Voici les différents canaux disponibles dans Reo [Kan, 2005]. On peut remarquer pour une grande partie des canaux une similitude avec le fonctionnement des circuits électriques.

Sync Un canal *Sync* a une source et une destination. Les données sont seulement transférées s'il y a simultanément une opération *write* sur la source et une opération *take* sur la destination. La figure 2.2 représente un canal *Sync*.

Figure 2.2 – Canal *Sync* [Kan, 2005]

Filter Le canal *Filter (P)* a une source et une destination. Le comportement de ce canal est identique au canal *Sync* à part qu'ici on vérifie avant si un certain pattern P est vrai. La figure 2.3 représente un canal *Filter*.

Figure 2.3 – Canal *Filter* [Kan, 2005]

SyncDrain Un canal *SyncDrain* est un canal où les deux extrémités du canal sont des sources. L'opération *write* peut avoir lieu sur les deux extrémités. Cependant, les opérations *write* peuvent seulement réussir simultanément. Toutes les données qui sont écrites sont perdues. La figure 2.4 représente un canal *SyncDrain*.

Figure 2.4 – Canal *SyncDrain* [Kan, 2005]

SyncSpout Le canal *SyncSpout* est l'opposé du *SyncDrain*. Au lieu des sources, les deux extrémités du canal sont des destinations. Les données sont envoyées si les deux côtés veulent effectuer simultanément une opération *take*. Les données qui sont transférées sont des données aléatoires. Il est possible d'appliquer un modèle tels que les données sont choisies parmi un ensemble restreint, par exemple des nombres à partir d'un intervalle de 1 à 10. La figure 2.5 représente un canal *SyncSpout*.

Figure 2.5 – Canal *SyncSpout* [Kan, 2005]

FIFO Un canal FIFO a une source et une destination, mais également un buffer illimité. L'extrémité du canal accepte toujours les données et les stocke dans son buffer. Les opérations *take* sur la destination réussissent seulement si des données sont disponibles dans le buffer. Les données sélectionnées dans le buffer sont disponibles dans l'ordre FIFO (First In, First Out) ce qui veut dire qu'elles sont enlevées dans le même ordre qu'elles sont rentrées dans le buffer. Un canal FIFO est asynchrone. La figure 2.6 représente un canal FIFO.

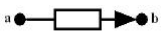

Figure 2.6 – Canal FIFO [Kan, 2005]

FIFOn Le canal FIFOn a, contrairement à FIFO, un buffer limité. Le nom indique la taille du buffer, par exemple FIFO1 a un buffer de taille 1. Un canal FIFO ou un canal FIFOn peut également être initialisé avec des données déjà disponibles dans le buffer.

LossySync Un canal *LossySync* a une source et une destination. La source accepte toujours une donnée, mais elle est seulement transférée à la destination si une opération *take* est présente. Il s'agit d'un canal synchrone sinon la donnée est perdue. La figure 2.7 représente un canal *LossySync*.

Figure 2.7 – Canal *LossySync* [Kan, 2005]

Join Le canal *Join* est employé pour construire des connecteurs plus complexes. On distingue trois types de nœuds : *source node*, *sink node* and *mixed nod*e.

source node : replicate Un *source node* est un nœud où seulement les extrémités du canal source coïncident. La figure 2.8 montre un *source node* où trois extrémités du canal source coïncident. Une opération *write* réussit seulement si toutes les extrémités du canal source acceptent la donnée. Il agit comme un réplicateur.

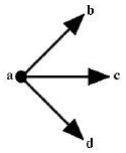

Figure 2.8 – *Source node* [Kan, 2005]

sink node : merge Un *sink node* est un nœud où seulement les extrémités du canal destination coïncident. La figure 2.9 montre un *sink node* où trois extrémités du canal destination coïcident. Une donnée est seulement transférée si simultanément une opération *take* et au moins une opération *write* ont lieu. Si des données sont transmises par de multiples destinations, alors l'un des canaux est choisi de manière non-déterministe. Il agit comme une fusion.

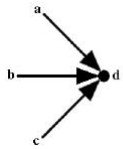

Figure 2.9 – *Sink node* [Kan, 2005]

mixed node Un *mixed node* se compose aussi bien de *source node* que de *sink node*. Son comportement sera donc un mélange des deux comportements. Des données sont transférées seulement si simultanément au moins un nœud de destination reçoit une donnée et tous les nœuds sources acceptent cette donnée. Si plusieurs nœuds sources veulent écrire, alors l'un d'entre eux est choisi de manière non-déterministe. Ainsi, un *mixed node* agit aussi bien comme un réplicateur qu'une fusion. La figure 2.10 représente un *mixed node*.

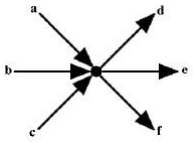

Figure 2.10 – *Mixed node* [Kan, 2005]

2.3 KLAVA : coordination par espace de tuples

KLAVA [Bettini *et al.*, 2002] est un framework Java basé sur le modèle de coordination KLAIM. KLAIM « Kernel Language for Agent Interaction and Mobility » est de De Nicola *et al.* C'est un modèle de coordination orienté données et basé sur la notion d'espaces de tuples.

Nous avons choisi d'utiliser KLAVA pour plusieurs raisons. Tout d'abord, nous avons trouvé une distribution récente du langage (2004) ce qui n'était pas le cas des autres langages que nous avons examinés (Linda[4]). De plus, KLAVA comme Reo avait l'avantage d'être écrit en Java et de bénéficier d'une distribution *open source*.

Nous avons utilisé la distribution KLAVA de Lorenzo Bettini[5].

2.3.1 Concepts

KLAIM est basé sur le modèle de coordination Linda. Linda est un langage de coordination avec des communications asynchrones et une mémoire partagée. La mémoire partagée est appelée un espace de tuples (*tuple space*) et contient des *tuples*.

Un tuple est une séquence d'informations appelée champs. Voici la création d'un tuple qui contient le nom de la variable (x) et son contenu (10).

```
new Tuple("x", 10);
```

La classe *TupleSpace* inclut des méthodes pour ajouter et supprimer des tuples de l'espace de tuples.

Voici la liste des méthodes possibles sur un espace de tuples.

in(t) Regarder pour un tuple t' qui correspond à t. Quand le tuple t' est trouvé, il est supprimé de l'espace de tuples. La valeur correspondant à t' est assignée aux champs de t et l'opération est terminée. S'il n'y a pas de correspondance, l'opération est suspendue jusqu'à ce qu'un tuple corresponde.

read(t) Identique à $in(t)$ mais ici le tuple t' n'est pas supprimé de l'espace de tuples.

out(t) Ajouter le tuple t dans l'espace de tuples.

Voici le code Java pour ajouter une variable dans l'espace de tuples.

```
TupleSpace TS = new TupleSpace() ;
TS.out(new Tuple("x", 10));
```

Pour récupérer notre variable x, il faut définir le type de données que nous cherchons. Par exemple, nous indiquons que nous cherchons un tuple avec comme premier paramètre une *string* contenant x et un entier comme second paramètre.

```
Tuple t = new Tuple("x", Integer.class);
TS.read(t) ;
```

[4]Linda de l'Université de Yale aux États-Unis date de 1986.

[5]Disponible sur le site : `http://music.dsi.unifi.it/download/` (visité le 20/08/2007).

2.4 Conclusion

Ce chapitre a présenté de façon concise un état de l'art des langages de coordination. Nous avons tout d'abord commencé par plusieurs définitions de ce qu'est la coordination ([Carriero et Gelernter, 1989], [Singh, 1992] et [Malone et Crowston, 1994]), puis nous avons enchaîné avec la définition d'un modèle de coordination ([Wegner, 1997]) et d'un language de coordination [Carriero et Gelernter, 1989]. Nous avons également présenté les deux grandes familles selon [Arbab et Talcott, 2002] : les orientés données (« *data-driven* ») et les orientés événements (« *control-driven* »). Enfin, nous nous sommes surtout concentrés sur les langages que nous avons utilisés dans la suite de ce mémoire. Nous avons choisi Reo ([Kan, 2005]) pour la coordination par événement et KLAVA ([Bettini *et al.*, 2002]) pour la coordination par espace de tuples.

Dans le chapitre suivant nous choisissons un fichier de modélisation.

Chapitre 3

Modélisation d'un workflow

Dans ce chapitre, nous analysons différents langages de modélisation d'un workflow et en choisissons un pour notre moteur de workflow. Une fois ce choix effectué, nous présentons ce langage. Finalement, nous reprenons notre exemple du premier chapitre pour le modéliser dans le langage de notre moteur de workflow.

3.1 Standards

Avant de commencer l'implémentation, nous devons choisir un format pour la modélisation de notre workflow. Les standards qui existent pour décrire les processus métiers sont très variés. La plupart sont basés sur XML (Extensible Markup langage).

Voici un aperçu des différents langages de modélisation disponibles :

XPDL - XML Process Definition langage [1] est « un standard de la Workflow Management Coalition qui permet de définir un processus métier ou processus d'affaires à l'aide du langage XML, processus métier qui sera ensuite utilisé par un moteur de workflow. »

UML - UML activity diagrams [2] est « une norme de l'OMG (Object Management Group). UML permet de modéliser des flux informatiques et d'organisation et est aussi de plus en plus utilisé pour modéliser des workflow. »

XLANG - XML Business Process langage [3] est « un standard de Microsoft. Il est une extension de WSDL (Web Services Description langage), le langage de définition des web services. Il fournit en même temps un modèle pour une orchestration des services et des contrats de collaboration entre ceux-ci. »

WSFL - Web Services Flow Langage [4] est « un standard de IBM. Il permet l'intégration (orchestration) des web services. Il se base et complète les outils existants comme SOAP (Simple Object Access Protocol), WSDL (Web Services Description Language), XMLP (XML Protocol) et UDDI (Universal Description Discovery and Integration). »

[1] `http://fr.wikipedia.org/wiki/XPDL` (visité le 09/11/2006).

[2] `http://is.tm.tue.nl/research/patterns/uml.htm` (visité le 09/11/2006).

[3] `http://www.alaide.com/dico.php?q=XLANG&ix=4843` (visité le 09/11/2006).

[4] `http://www.bpms.info/lexique_detail.asp?ref=268` (visité le 09/11/2006).

BPEL - Business Process Execution langage [5] est « une combinaison de WSFL et XLANG. Il a été lancé par IBM, Microsoft et BEA. C'est un langage de programmation basé sur XML et permettant de définir une tâche par la combinaison de web services. BPEL4WS utilise WSDL pour décrire les actions d'un processus. »

WSCI - Web Service Choregraphy Interface [5] est « un standard de IBM. C'est une norme d'orchestration de web services qui décrit le flux de messages échangés par un web service lors de son interaction avec d'autres services. L'orchestration, appelée parfois chorégraphie (« *choregraphy* »), assure la succession des tâches, le contrôle de la bonne exécution, les reprises en cas d'incident, etc. »

BPML - Business Process Modelling Langage [6] est « un standard de BPMI (Business Process Management Initiative). Il est complémentaire à WSCI. »

Dans le tableau 3.1, nous vérifions l'expressivité de chacun des langages présentés précédement. Pour ce faire, on vérifie, pour chaque pattern de worflow, s'il peut être implémenté dans ce langage. Ce travail a été réalisé par [van der Aalst *et al.*, 2003]. Si le standard supporte le pattern dans une de ces constructions, il est évalué « + ». S'il n'est pas directement supporté, il est évalué « +/- ». N'importe quelle solution qui donne un diagramme ou du code spaghetti est considérée comme ne le supportant pas et est évaluée « - » car des workflow trop compliqués deviennent illisibles et difficilement maintenables. Notons qu'un modèle est seulement soutenu directement s'il y a un dispositif fourni par le langage qui soutient la construction sans recourir à n'importe laquelle des solutions mentionnées dans l'implémentation du pattern.

Notre choix de structure s'est dès lors porté sur YAWL vu que c'est lui qui implémente le plus grand nombre de patterns de workflow de manière directe. Le seul pattern qui n'est pas implémenté directement est le pattern « *Implicit Termination* ». Le concepteur est forcé d'identifier un nœud final unique. Un modèle avec des nœuds de fin multiples peut être transformé en un modèle avec un nœud de fin unique. En fait, cette restriction a été implémentée de façon explicite afin que tout workflow ne contienne toujours qu'un seul nœud de fin.

3.2 Modélisation

Dans cette section, nous expliquons la modélisation utilisée dans YAWL
[van der Aalst et Hofstede, 2002].

La spécification de workflow dans YAWL se compose d'un ensemble de « *process definition* » qui forment une hiérarchie. Cette hiérarchie est équivalente à la structure bien connue d'arbre. Chaque *process definition* se compose de tâches (« *task* ») et d'états (« *conditions* »). Une tâche peut être *atomique* ou *composée*. Chaque *process definition* a une seule condition d'entrée (*input condition*) et de sortie (*output condition*). Une tâche composée est une tâche qui fait référence à un *process definition* à un niveau plus bas dans l'arbre. Une tâche atomique et un état sont tout simplement une feuille de l'arbre. Chaque tâche peut être instanciée plusieurs fois (« *Multiple instances of an atomic task* » et « *Multiple instances of a composite task* »). Un état (*conditon*) permet de présenter l'état au sein d'un processus.

[5] `http://www.guideinformatique.com/fiche-orchestration_processus_xlang_wsfl_bpel4ws_xpdl-380.html` (visité le 09/11/2006).

[6] `http://en.wikipedia.org/wiki/Business_Process_Execution_Language` (visité le 09/11/2007).

Pattern	XPDL	UML	BPEL	XLANG	WSFL	BPML	WSCI	YAWL
Sequence	+	+	+	+	+	+	+	+
Parallel Split	+	+	+	+	+	+	+	+
Synchronization	+	+	+	+	+	+	+	+
Exclusive Choice	+	+	+	+	+	+	+	+
Simple Merge	+	+	+	+	+	+	+	+
Multi Choice	+	-	+	-	+	-	-	+
Synchronizing Merge	+	-	+	-	+	-	-	+
Multi Merge	-	-	-	-	-	+/-	+/-	+
Discriminator	-	-	-	-	-	-	-	+
Arbitrary Cycles	+	-	-	-	-	-	-	+
Implicit Termination	+	-	+	-	+	+	+	-
MI without Synchronization	+	-	+	+	+	+	+	+
MI with a Priori Design Time Knowledge	+	+	+	+	+	+	+	+
MI with a Priori Runtime Knowledge	-	+	-	-	-	-	-	+
MI without a Priori Runtime Knowledge	-	-	-	-	-	-	-	+
Deferred Choice	-	+	+	+	-	+	+	+
Interleaved Parallel Routing	-	-	+/-	-	-	-	-	+
Milestone	-	-	-	-	-	-	-	+
Cancel Activity	-	+	+	+	+	+	+	+
Cancel Case	-	+	+	+	+	+	+	+

Tab. 3.1 – Comparaison des standards de modélisation [van der Aalst *et al.*, 2003]

Le « *top-level workflow* » est le seul *process definition* qui n'est pas référencé par une tâche composée. Il forme donc la racine de la structure en arbre. YAWL dispose aussi d'une notation pour supprimer des jetons d'une région. Cette notation est utile pour les patterns d'annulation (voir 5.4.6).

Enfin, voici les connecteurs disponibles dans YAWL :

AND-split task une tâche qui se divise en deux tâches ;

AND-join task deux tâches qui se regroupent en une tâche ;

XOR-split task une tâche qui se divise en deux tâches mais le flux ne transite que dans l'une des deux tâches ;

XOR-join task deux tâches qui se regroupent en une tâche mais seul le flux d'une des deux transitera vers la tâche ;

OR-split task une tâche qui se divise en deux tâches et le flux transite dans une des deux tâches ou dans les deux ;

OR-join task deux tâches qui se regroupent en une tâche et le flux d'une ou des deux transitera vers la tâche.

La figure 3.1 [van der Aalst et Hofstede, 2002] montre les différents éléments qui peuvent être utilisés lors de la modélisation d'un workflow dans YAWL et que nous avons expliqué dans cette section.

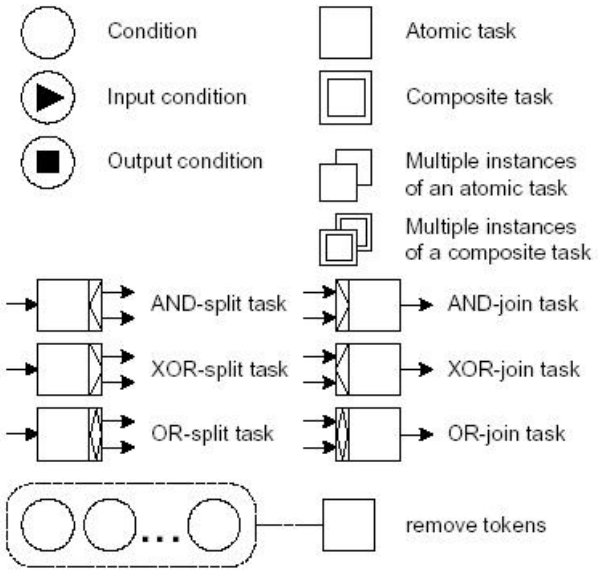

Figure 3.1 – Éléments de YAWL [van der Aalst et Hofstede, 2002]

Finalement, il est possible d'indiquer une limite inférieure et une limite supérieure pour le nombre d'instances créés après lancement de la tâche. Il est aussi possible d'indiquer que la tâche se termine au moment où un certain nombre d'instances sont finies. Au moment où ce seuil est atteint, toutes les instances courantes sont terminées et la tâche est terminée. Si aucun seuil n'est indiqué, la tâche est terminée dès que toutes les instances sont terminées. Enfin, il y a un quatrième paramètre indiquant si le nombre d'instances est fixe après avoir créé les instances initiales. La valeur du paramètre est « statique » si après création aucune instance ne peut être ajoutée et « dynamique » s'il est possible d'ajouter des instances pendant l'exécution de la tâche.

3.3 Exemple

Ici, nous reprenons l'exemple de workflow présenté au premier chapitre. Pour rappel, cet exemple modélise la préparation d'un voyage par une agence de voyage. La figure 3.2 montre ce workflow en YAWL.

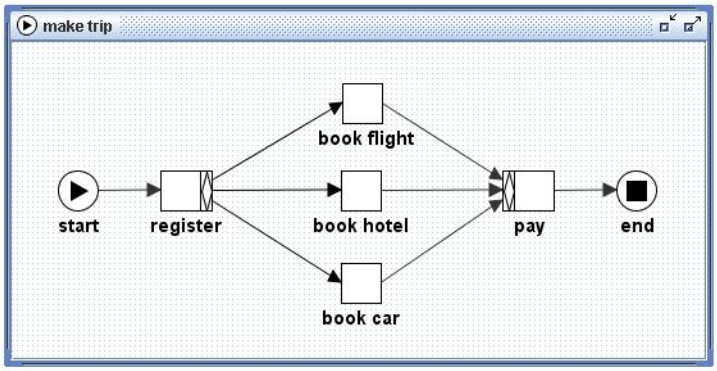

Figure 3.2 – Modélisation de l'exemple 1 avec l'éditeur YAWL

Un voyage peut se composer de plusieurs étapes. C'est pourquoi, nous raffinons notre workflow en introduisant un sous-processus « *do iteneray segment* ». Ce sous-processus est instancié pour chaque étape. La figure 3.3 montre ce workflow en YAWL.

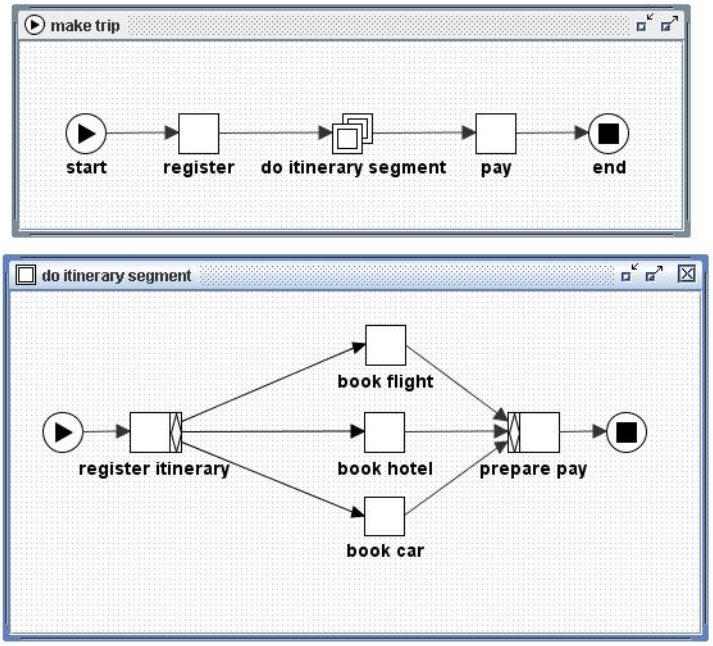

Figure 3.3 – Modélisation de l'exemple 2 avec l'éditeur YAWL

3.4 Conclusion

Nous venons de présenter une étude comparative sur les différentes modélisations existantes pour un processus métier [van der Aalst *et al.*, 2003]. Cette étude montre les points positifs et négatifs de ces différents formats. Celle-ci nous a permis de choisir notre format de modélisation (YAWL). Ensuite, nous avons présenté la structure du fichier et les différents composants pour construire un workflow [van der Aalst et Hofstede, 2002]. Enfin, nous avons modélisé notre exemple de workflow avec l'éditeur YAWL.

Dans le chapitre suivant nous présentons l'analyse de la perspective de contrôle de flux.

Chapitre 4

Perspective de contrôle de flux - Analyse

Dans ce chapitre, nous analysons le flux de contrôle au sein du moteur de workflow. Cette perspective a pour but de décrire les tâches et l'ordre d'exécution. Ici, nous nous limitons explicitement à cette perspective et nous ignorons donc volontairement tous les autres aspects qui sont liés aux autres perspectives (données, ressources et opérationnelles). Nous commençons avec le parsing du fichier XML de modélisation. De ces informations, nous modélisons le domaine de l'application. Ensuite, nous établissons les scénarios d'utilisation, les diagrammes d'états et les diagrammes de séquencement.

4.1 Parsing du fichier XML

Dans cette section, nous présentons le fichier XML que le moteur de workflow reçoit en entrée. Nous expliquons sa structure ainsi que les informations que nous extrayons et qui seront utilisées lors de l'analyse.

Le fichier XML décrit le processus métier que le moteur de workflow va devoir créer. Le pattern « Composite » [Gamma *et al.*, 1995] est utilisé pour décrire un processus. Ici, un processus est composé d'une ou plusieurs tâches et un processus est lui-même une tâche composée ayant la particularité d'avoir un attribut *root*.

Nous avons modélisé la structure d'une tâche composée sur la figure 4.1. Le fichier XML pour notre exemple de workflow présenté au premier chapitre et modélisé avec YAWL dans le chapitre précédent se trouve en annexe B.

Une tâche est décrite sous le tag *decomposition*. On a autant de tags *decomposition* qu'il y a de tâches atomiques et de tâches composées dans notre workflow. Nous avons comme attribut un identifiant et un type de tâche.

Une tâche atomique contient encore d'autres propriétés qui sont pertinentes mais pour les autres perspectives.

Une tâche composée contient un tag *processControlElements* afin de décrire le flux au sein d'une tâche composée. Cette section est structurée en quatre parties qui sont l'*input condition*, la ou les *task*, la ou les *condition* et l'*output condition*. Chacunes de ces parties ont comme attribut un identifiant.

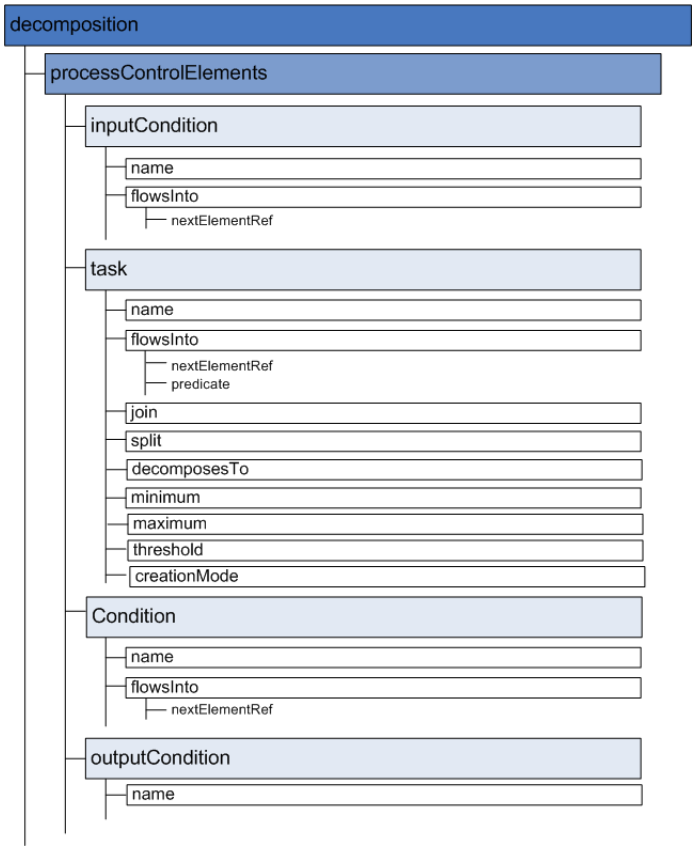

Figure 4.1 – Structure du fichier XML

L'*input condition* est le point d'entrée de la tâche composée. Elle est composée d'un nom et du ou des élément(s)[1] suivant(s).

La *task* est la description du flux d'une tâche au sein de la tâche composée. On va retrouver autant de *task* qu'il y a de tâches dans la tâche composée. Elle est composée d'un nom, du ou des élément(s) suivant(s), du type de join (*join*) et du type de split (*split*). Dans le cas d'une tâche composée, il y a également la référence vers la description de cette tâche composée (*decomposeTo*). Dans le cas d'instances multiples, elle est en plus composée du minimum, maximum, le seuil (*threshold*) et le mode de création pour les instances (*createMode*).

La *condition* est la description d'un état au sein du processus. Elle n'est pas obligatoire dans une tâche composée. Elle est composée d'un nom et du ou des élément(s) suivant(s).

L'*output condition* est le point de sortie de la tâche composée. Elle est composée uniquement du nom.

Le tag *flowsInto* permet d'indiquer la ou les éléments(s) suivant(s) avec la possibilité d'indiquer une condition sur le choix de l'embranchement (*predicate*). Les éléments sont identifiés par l'identifiant unique au sein du workflow. Cet identifiant est placé dans *nextElementRef*.

Notons que nous considérons que le fichier XML reçu est valide. La structure du fichier XML est validée à l'aide d'un schéma XML. La sémantique du fichier XML est vérifiée par « *YAWL designer* » c'est-à-dire qu'on ne peut pas recevoir un fichier XML incohérent. Par exemple, un nombre négatif d'instances de tâches, ou encore deux tâches qui ont des identifiants identiques.

Un schéma XML de notre fichier de modélisation se trouve en annexe A.

[1] Elément = tâche (*task*) ou état (*condition*)

4.2 Modélisation du domaine d'application

Le diagramme de classes de la figure 4.2 montre la modélisation de la perspective de flux de contrôle. Nous rappelons que ce modèle se limite à cette perspective. Nous le complétons dans les chapitres suivants.

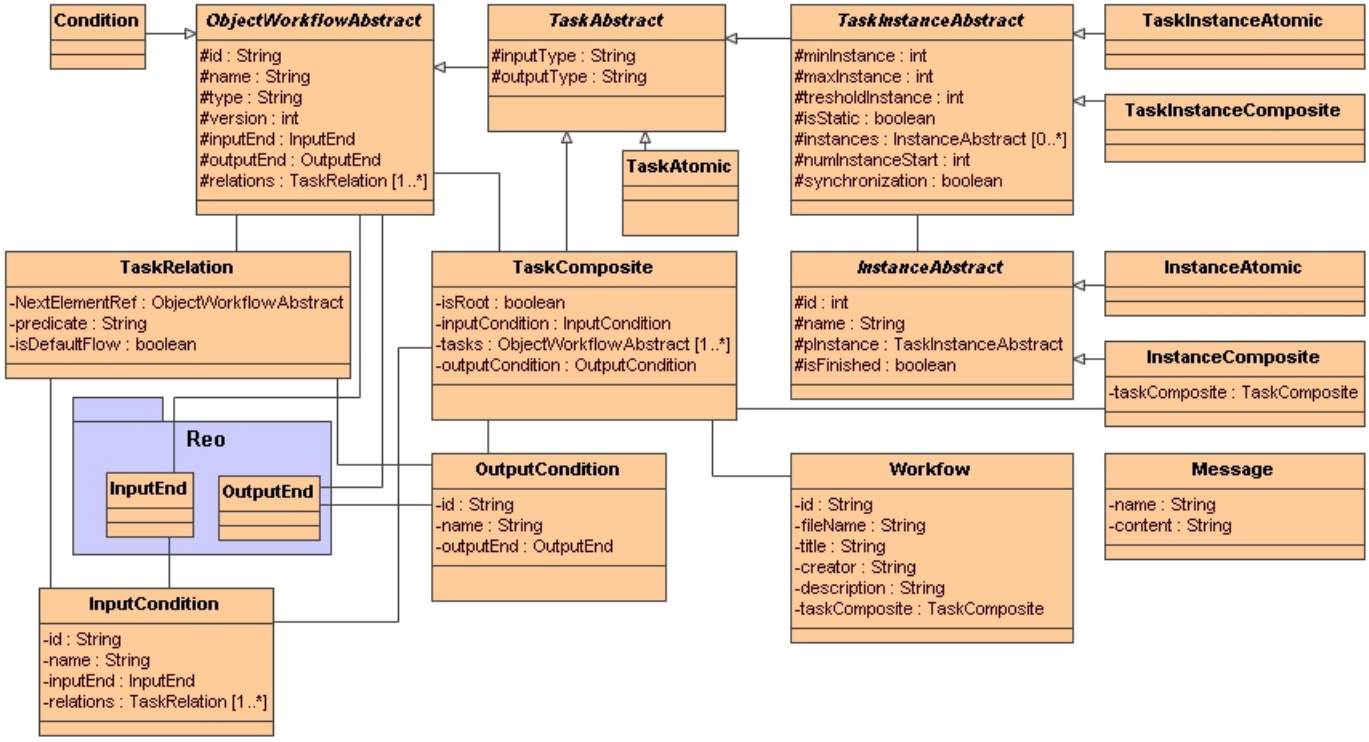

Figure 4.2 – Diagramme de classes

Workflow

La classe WORKFLOW représente le contenu du fichier de modélisation du workflow. Cette classe contient :

id L'identifiant du workflow.

fileName Le nom du fichier.

title Le titre du processus métier.

creator Le nom du créateur.

description La description du processus métier.

taskComposite La tâche composée qui constitue le workflow.

ObjectWorkflowAbstract

La classe abstraite ObjectWorkflowAbstract représente tous les attributs communs d'une tâche et d'un état. Elle ne sera pas présente en tant que telle dans le workflow mais est la généralisation d'une tâche atomique, d'une tâche composée et d'un état. Cette classe contient :

id La clé unique pour représenter un objet du workflow

name Le nom de l'objet.

type Le type d'objet.

inputEnd Le canal de sortie de l'objet.

outputEnd Le canal d'entrée de l'objet.

relations La ou les relations entre les éléments.

TaskAbstract

La classe abstraite TaskAbstract représente tous les attributs communs d'une tâche. Elle ne sera pas présente en tant que telle dans le workflow mais est la généralisation d'une tâche atomique et d'une tâche composée. Cette classe hérite de tous les attributs de la classe abstraite ObjectWorkflowAbstract et contient en plus :

inputType Le type de connecteur en entrée d'une tâche. Par exemple : AndJoin.

outputType Le type de connecteur à la sortie d'une tâche. Par exemple : AndSplit.

TaskAtomic

La classe TaskAtomic représente une tâche atomique du workflow. Cette classe hérite de tous les attributs de la classe abstraite TaskAbstract et ne contient pas d'autres attributs. Ce sera donc une instanciation concrète de la classe abstraite.

TaskComposite

La classe TaskComposite représente une tâche composée du workflow. Cette classe hérite de tous les attributs de la classe abstraite TaskAbstract et contient en plus :

isRoot Un booléen qui indique la racine du workflow.

inputCondition L'entrée de la tâche composée.

tasks Un tableau qui représente les différentes tâches de la tâche composée.

outputCondition La sortie de la tâche composée.

InputCondition

La classe InputCondition représente l'entrée dans une tâche composée.

id La clé unique pour représenter un objet du workflow.

name Le nom de l'objet qui représente l'entrée dans une tâche composée.

inputEnd Le canal d'entrée d'une tâche composée.

relations La ou les relations entre l'entrée dans une tâche composée et les sous-composants.

OutputCondition

La classe OutputCondition représente la sortie dans une tâche composée.

id La clé unique pour représenter un objet du workflow.

name Le nom de la sortie dans une tâche composée.

outputEnd Le canal de sortie d'une tâche composée.

Condition

La classe Condition représente un état du workflow. Cette classe hérite de tous les attributs de la classe abstraite ObjectWorkflowAbstract et ne contient pas d'autres attributs. Ce sera donc une instanciation concrète de la classe abstraite.

TaskRelation

La classe TaskRelation représente le successeur de chaque tâche.

nextElementRef L'élément suivant.

predicat La condition pour emprunter cette relation.

isDefaultFlow Un booléen qui indique le chemin choisi par défaut par le moteur de workflow.

Message

La classe Message représente le message qui va circuler entre deux tâches.

Name L'identifiant de l'objet qui envoit le message.

Content Le contenu du message.

Elle contient également 3 constantes afin d'avertir la tâche suivante qu'on ne se trouve pas dans un cas normal d'exécution :
NOTASK : la tâche ne doit pas être exécutée ;
CANCELTASK : la tâche est annulée en cours d'exécution ;
CANCELWORKFLOW : l'instance de workflow est annulée en cours d'exécution.

TaskInstanceAbstract

La classe abstraite TaskInstanceAbstract représente tous les attributs communs pour la gestion des instances d'une tâche. Elle ne sera pas présente en tant que telle dans le workflow mais est la généralisation d'une instance de tâche atomique et d'une instance de tâche composée.

minInstance Le nombre minimum d'instances.

maxInstance Le nombre maximum d'instances.

thresholdInstance Le seuil du nombre d'instances attendues pour terminer la tâche.

isStatic Les instances de tâche peuvent être créés soit dynamiquement, soit statiquement.

instances Un tableau qui reprend toutes les instances qui ont été créées pour cette tâche.

numInstanceStart Le nombre d'instances qui sont créées au démarrage de la tâche.

synchronization Le tâche attend ou pas ces instances pour se terminer.

TaskInstanceAtomic

La classe TaskInstanceAtomic représente la gestion des instances d'une tâche atomique. Cette classe hérite de tous les attributs de la classe abstraite TaskInstanceAbstract et ne contient pas d'autres attributs. Ce sera donc une instanciation concrète de la classe abstraite.

TaskInstanceComposite

La classe TaskInstanceComposite représente la gestion des instances d'une tâche composée. Cette classe hérite de tous les attributs de la classe abstraite TaskInstanceAbstract et ne contient pas d'autres attributs. Ce sera donc une instanciation concrète de la classe abstraite.

InstanceAbstract

La classe abstraite InstanceAbstract représente tous les attributs communs d'une instance de tâche. Elle ne sera pas présente en tant que telle dans le workflow mais est la généralisation d'une instance de tâche atomique et d'une instance de tâche composée.

id La clé unique pour représenter une instance de tâche.

name Le nom de l'instance de tâche.

mInstance Une référence vers TaskInstanceAbstract pour pouvoir utiliser la méthode *write* une fois que l'instance est terminée.

isFinished Un booléen qui indique que l'instance est terminée.

InstanceAtomic

La classe InstanceAtomic représente une instance d'une tâche atomique. Cette classe hérite de tous les attributs de la classe abstraite InstanceAbstract et ne contient pas d'autres attributs. Ce sera donc une instanciation concrète de la classe abstraite.

InstanceComposite

La classe InstanceComposite représente une instance d'une tâche composée. Cette classe hérite de tous les attributs de la classe abstraite InstanceAbstract et contient en plus :

compositeTask La tâche composée qu'il doit effectuer.

4.3 Scénario d'utilisation

Nous présentons les différents scénarios d'utilisation entre l'utilisateur et le moteur de workflow pour la perspective de contrôle de flux.

Workflow

Le scénario d'utilisation de la figure 4.3 montre les différentes possibilités que l'utilisateur *Admin* a à sa disposition pour gérer le workflow. Un administrateur peut donc charger un workflow en mémoire (*Load workflow*), annuler un workflow (*Cancel workflow*) et visualiser l'état du workflow (*View workflow*).

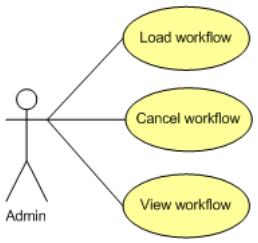

Figure 4.3 – Scénario d'utilisation d'un workflow

Instance d'un workflow

Dans cette section, le scénario d'utilisation de la figure 4.4 montre les différentes possibilités que l'utilisateur *Admin* a à sa disposition pour gérer les instances d'un workflow. Un administrateur peut donc lancer une instance de workflow (*Start instance workflow*), réinitialiser une instance de workflow (*Reset instance workflow*) et annuler une instance de workflow (*Cancel instance workflow*).

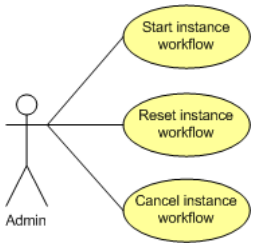

Figure 4.4 – Scénario d'utilisation d'une instance de workflow

Tâche

Le scénario d'utilisation de la figure 4.5 montre les différentes possibilités que l'utilisateur *Admin* a à sa disposition pour gérer une tâche. Un administrateur peut donc éditer une tâche (*Edit task*), mettre la tâche en attente (*Pause task*), relancer une tâche en attente (*Resume task*), annuler une tâche (*Cancel task*) et réinitialiser une tâche (*Reset task*).

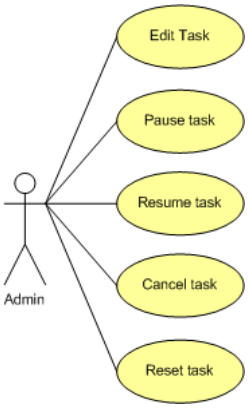

Figure 4.5 – Scénario d'utilisation d'une tâche

Instance de tâche

Le scénario d'utilisation de la figure 4.6 montre les différentes possibilités que l'utilisateur *Admin* a à sa disposition pour gérer les instances d'une tâche. Un administrateur peut donc ajouter une instance à une tâche (*Add instance*). Certains événements associés à une tâche ont des répercussions sur les instances. Par exemple mettre une tâche en attente a pour effet de mettre la ou les instance(s) de tâche en attente (*Pause instance task*) et relancer une tâche a pour effet de relancer la ou les instance(s) d'une tâche en attente (*Resume instance task*).

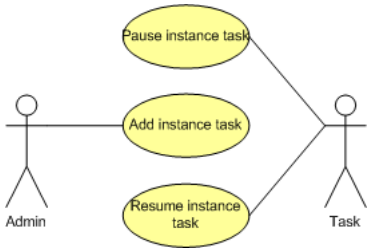

Figure 4.6 – Scénario d'utilisation d'une instance de tâche

4.4 Diagramme d'états

Nous présentons des diagrammes d'états afin de pouvoir comprendre les différentes étapes par lesquelles un workflow, une instance de workflow, une tâche et une instance de tâche peuvent passer.

Workflow

Le diagramme d'états de la figure 4.7 montre les différents états par lesquels un workflow peut passer. Le workflow est créé en appelant *load workflow* et son état suivant est *Workflow loaded*.

Une fois que le workflow est chargé en mémoire, on va pouvoir lancer des instances de ce workflow en appelant la méthode *start instance workflow* et on arrive dans l'état *Workflow running*. On reviendra à l'ancien statut une fois qu'il n'y a plus d'instance en cours d'exécution (*# instance = 0*).

Quand on est dans l'état *Workflow loaded* ou *Workflow running* on peut annuler le workflow en le supprimant et en appelant la méthode *cancel workflow*.

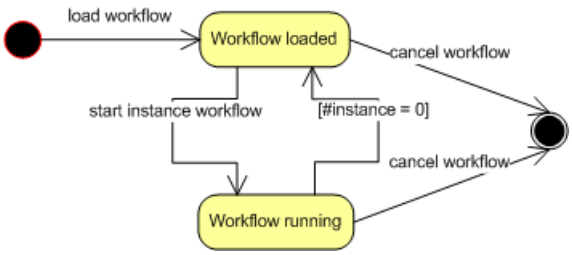

Figure 4.7 – Diagramme d'états d'un workflow

Instance d'un workflow

Le diagramme d'états de la figure 4.8 montre les différents états par lequels une instance de workflow peut passer. L'instance de workflow est créée en appelant *start instance workflow* et son état suivant est *Instance workflow running*.

Une fois que l'instance de workflow est en cours d'exécution :
- soit on suspend l'instance de workflow un moment en appelant *pause instance workflow* et on arrive à l'état *Instance workflow paused*. Pour la réactiver, on appelle *resume instance workflow* et on revient à l'état *Instance workflow running* ;
- soit on réinitialise l'instance de workflow en appelant *reset instance workflow* et on revient au même état qui est *Instance workflow running* ;
- soit l'instance du workflow est finie (*endInstanceTask*).

Quand on est dans l'état *Instance workflow running* ou *Instance workflow paused* on peut annuler l'instance de workflow en appelant la méthode *cancel instance workflow*.

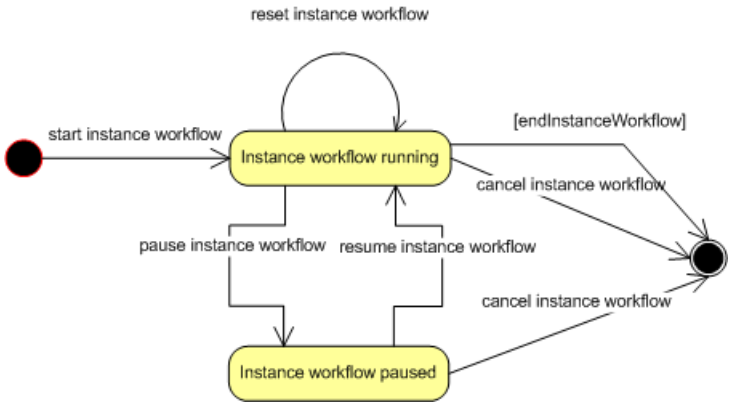

Figure 4.8 – Diagramme d'états d'une instance de workflow

Tâche

Le diagramme d'états de la figure 4.9 montre les différents états par lesquels une tâche peut passer.

La création d'une tâche s'effectue lors de l'appel au constructeur de la classe taskAtomic ou taskComposite et on arrive dans l'état *Task created*. Une fois qu'on a lancé la tâche on arrive dans l'état *task running*.

Dès que la tâche est en cours d'exécution :
- soit on suspend la tâche en appelant *pause task* et on arrive à l'état *Task paused*. Pour la réactiver, on appelle *resume task* et on revient à l'état *Task running* ;
- soit on réinitialise la tâche en appelant *reset task* et on revient à l'état *Task created* ;
- soit la tâche est finie (*endTask*).

Quand on est dans l'état *Task created* ou *Task running* ou *Task paused* on peut annuler le workflow en appelant la méthode *cancel task*.

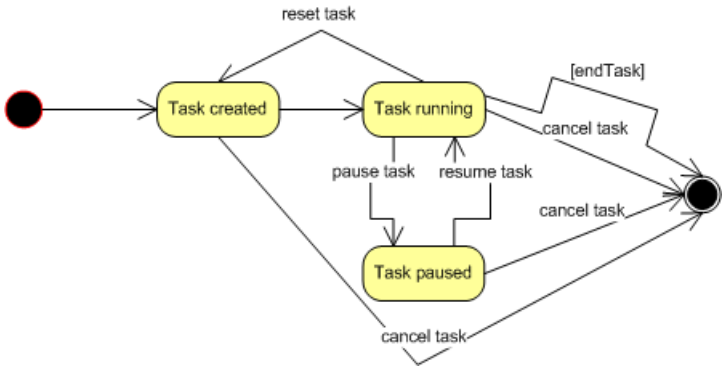

Figure 4.9 – Diagramme d'états d'une tâche

Instance de tâche

Le diagramme d'états de la figure 4.10 montre les différents états par lesquels une tâche et ses instances peuvent passer.

La création d'une tâche s'effectue lors de l'appel au constructeur de la classe taskAtomic ou taskComposite et on arrive dans l'état *Task created*.

On passera de l'état *task created* à *task and instance running* une fois qu'on aura atteint le nombre minimum d'instances nécessaires pour commencer la tâche ($\#instance = min$).

On peut ajouter des instances à partir de l'état
- *task created* si on n'a pas atteint le nombre minimum d'instances pour commencer la tâche ($\#instance < min$) ;
- *task running* si on est en mode *addDynamic* et qu'on n'a pas atteint le nombre maximum d'instances pour cette tâche ($\#instance < max$).

Une fois que la tâche et les instances sont en cours d'exécution :
- soit on suspend la tâche en appelant *pause task* et on arrive à l'état *Task and instance paused*. Pour la réactiver, on appelle *resume task* et on revient à l'état *Task and instance running* ;
- soit on réinitialise la tâche en appelant *reset task* et on revient à l'état *Task created* ;
- soit la tâche est finie.

La tâche est terminée :
- soit quand la tâche est finie si on est en mode non synchrone et les instances peuvent se terminer indépendement par la suite (*endTask* & *no synchronization*) ;
- soit quand toutes les instances sont finies si on est en mode synchrone (*synchronization* & *all instance is finished*) ;
- soit quand le nombre d'instances finies est égal au seuil ($\#instancefinished = \#threshold$) .

Quand on est dans l'état *Task created* ou *Task running* ou *Task paused* on peut annuler le workflow en appelant la méthode *cancel task*.

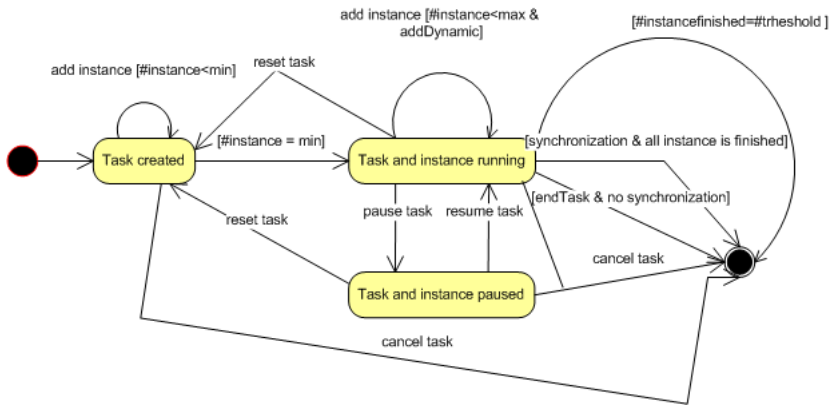

Figure 4.10 – Diagramme d'états d'une instance de tâche

4.5 Diagramme de séquencement

Nous présentons des diagrammes de séquencement afin de montrer comment se passe le chargement d'un workflow en mémoire, le lancemment d'une instance de workflow et l'annulation d'une instance de workflow. Nous négligeons volontairement certains aspects techniques afin de garder la clarté du diagramme.

Charger un workflow en mémoire

Le diagramme de séquencement de la figure 4.11 montre comment on charge un workflow en mémoire.

1. L'utilisateur *Admin* va demander le chargement d'un workflow et il va donner au système un fichier XML qui représente la modélisation du workflow en YAWL.
2. La classe Workflow va demander la création d'une tâche composée et donne le fichier XML en paramètre.
3. La classe TaskComposite crée l'entrée de la tâche composée.
4. La classe InputCondition renvoie l'objet qui vient d'être créé.
5. La classe TaskComposite crée la sortie de la tâche composée.
6. La classe OutputCondition renvoie l'objet qui vient d'être créé.
7. La classe TaskComposite va créer autant de tâches qu'il y a de sous-tâches dans la tâche composée. Chaque tâche est soit une tâche atomique soit une tâche composée. Pour un soucis de clarté nous englobons les deux dans task. Si c'est une tâche atomique, aucune autre classe n'est nécessaire et si c'est une tâche composée nous recommençons le même scénario. Le même mécanisme est utilisé pour les états mais au lieu d'appeler la classe task nous appelerons la classe condition. Nous ne l'indiquons pas dans le diagramme afin de ne pas l'allourdir.
8. La classe Task renvoie la tâche.
9. La classe TaskComposite va assigner les canaux pour l'entrée de la tâche composée.
10. La classe InputCondition renvoie l'objet et ses canaux.
11. La classe TaskComposite va assigner pour chaque tâche les canaux en entrée et en sortie. Même chose pour les conditions mais avec la classe condition.
12. La classe Task renvoie la tâche et ses canaux.

13. La classe TaskComposite renvoie la tâche composée.
14. La classe Workflow renvoie le workflow afin de mettre à jour l'écran de l'utilisateur.

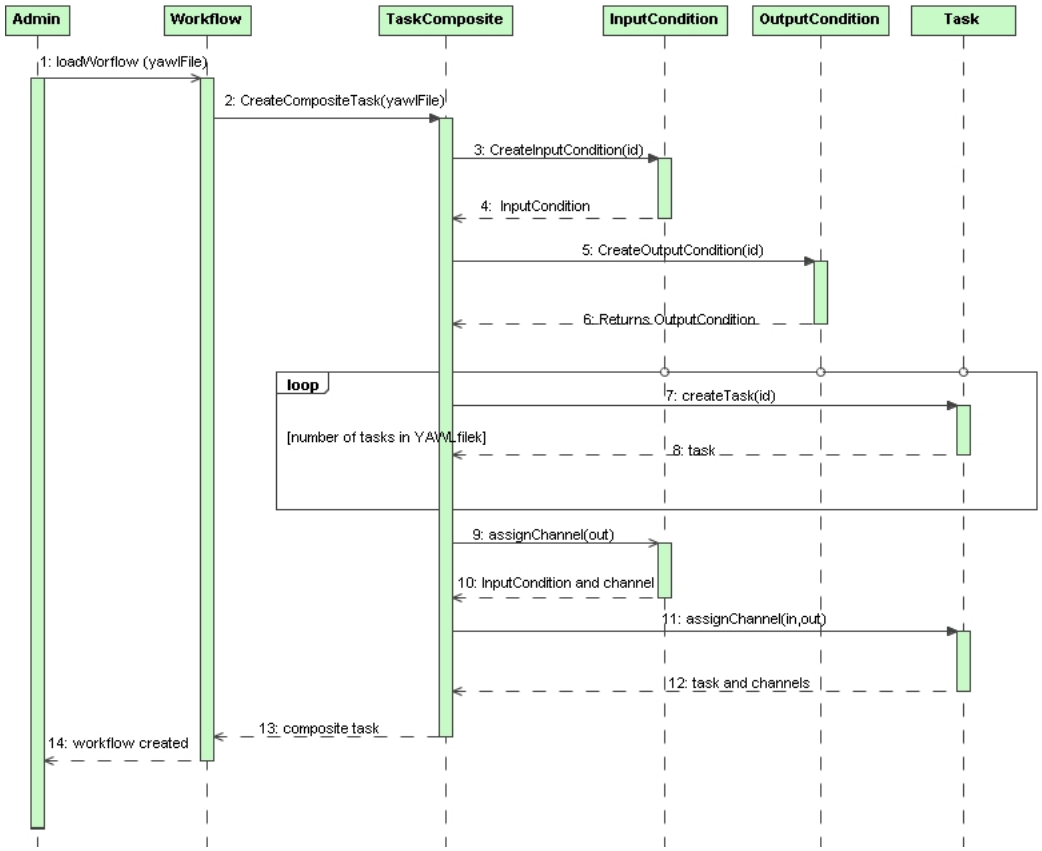

Figure 4.11 – Diagramme de séquencement - Charger un workflow en mémoire

Lancer une instance de workflow

Le diagramme de séquencement de la figure 4.12 montre comment lancer une instance de workflow.

1. L'utilisateur *Admin* va demander le lancement d'une instance d'un workflow et il va donner au système l'identifiant de ce workflow.
2. La classe Workflow va copier la tâche composée.
3. La classe TaskComposite renvoie la copie de la tâche composée.
4. La classe Workflow va lancer la tâche composée afin de lancer une nouvelle instance du workflow.
5. La classe TaskComposite va lancer récursivement chacune des tâches qui la composent. Cette tâche est soit une tâche atomique soit une tâche composée. Comme pour le chargement d'un workflow nous englobons les deux dans *task*.
6. La classe Task renvoie la tâche en cours d'exécution.
7. La classe TaskComposite renvoie la tâche composée en cours d'exécution.
8. La classe Workflow renvoie l'instance de workflow en cours d'exécution afin de mettre à jour l'écran de l'utilisateur.

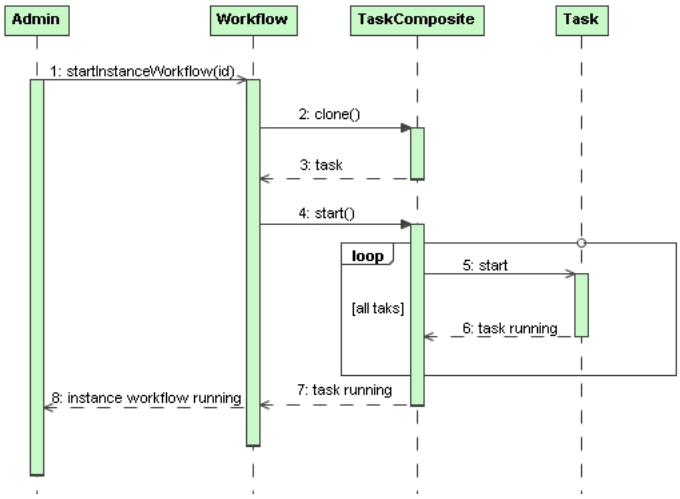

Figure 4.12 – Diagramme de séquencement - Lancer une instance de workflow

Annulation d'une instance de workflow

Le diagramme de séquencement de la figure 4.13 montre l'annulation d'une instance de workflow.

1. L'utilisateur *Admin* va demander l'annulation d'une instance de workflow et il va donner au système l'identifiant de ce workflow.
2. La classe Workflow va annuler la tâche composée.
3. La classe Task annule récursivement toutes les tâches en cours d'exécution qui la composent. Elle renvoie pour chacune de ses tâches un message d'annulation du workflow à la ou les tâche(s) suivante(s) afin de terminer proprement le workflow.
4. La classe Task renvoie *true* une fois que c'est fini sinon elle renvoie une exception.
5. La classe Workflow renvoie *true* une fois que c'est fini sinon elle renvoie une exception.

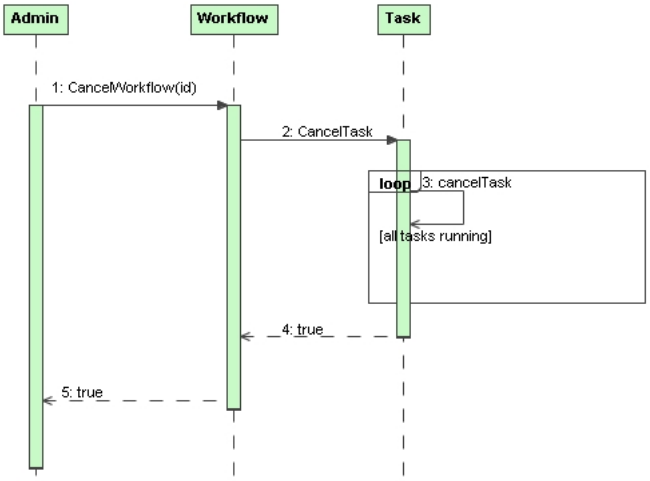

Figure 4.13 – Diagramme de séquencement - Annulation d'une instance de workflow

4.6 Conclusion

Nous venons d'analyser la perspective de contrôle de flux. Le point de départ de cette analyse est le fichier XML de modélisation. Nous avons également examiné les fonctionnalités de différents workflow pour établir les scénarios d'utilisation et les diagrammes d'états. Enfin, les diagrammes de séquencement se basent sur notre analyse et sur la technologie que nous avons choisie.

Dans le chapitre suivant, nous présentons l'implémentation de cette analyse.

Chapitre 5

Perspective de contrôle de flux - Implémentation

Dans ce chapitre, nous implémentons la perspective de contrôle de flux au sein du moteur de workflow. Nous commençons par l'implémentation des connecteurs, des tâches, des instances de tâches et des états. Nous montrons que notre implémentation est correcte grâce à des patterns de workflow pour le contrôle de flux.

5.1 Connecteurs

Cette section explique de façon détaillée comment nous implémentons les six connecteurs utilisés lors de la modélisation d'un workflow. Chaque connecteur est présenté via sa représentation en Reo puis via son implémentation en Java. Le code complet est disponible en annexe C.1.

5.1.1 AND-Split

Le connecteur *AND-Split* permet à un canal de se diviser en deux canaux.

Modélisation Reo

La figure 5.1 représente le connecteur *AND-Split* en Reo.

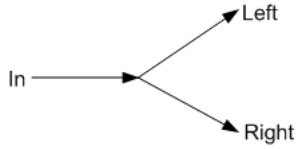

Figure 5.1 – Connecteur AND-Split en Reo

Implémentation Java

Ce connecteur est déjà implémenté dans Reo et se nomme *Replicator*. Notre connecteur a une entrée (*in*) et deux sorties (*left* et *right*). Voici le corps de notre classe Java.

```
Replicator r = new Replicator();
in = r.getIn();
left = r.getLeft();
right = r.getRight();
```

5.1.2 AND-Join

Le connecteur *AND-Join* permet à deux canaux de se regrouper en un canal.

Modélisation Reo

La figure 5.2 représente le connecteur *AND-Join* en Reo.

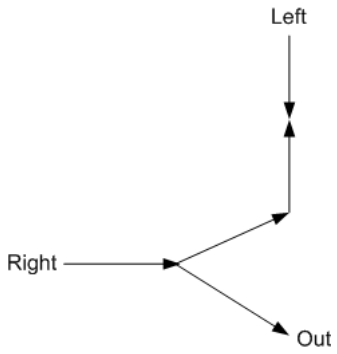

Figure 5.2 – Connecteur AND-Join en Reo

Implémentation Java

Nous utilisons les connecteurs *Replicator* et *SyncDrain* qui sont déjà implémentés dans Reo. Notre connecteur a deux entrées (*left* et *right*) et une sortie (*out*). Voici le corps de notre classe Java.

```
SyncDrain sd = new SyncDrain();
Replicator replicator = new Replicator();
sd.getRight().JOIN(replicator.getLeft());
left = sd.getLeft();
right = replicator.getIn();
out = replicator.getRight();
```

5.1.3 XOR-Split

Le connecteur *XOR-Split* permet à un canal de se diviser en deux canaux mais le flux ne transite que dans l'un des deux canaux.

Modélisation Reo

La figure 5.3 représente le connecteur *XOR-Split* en Reo.

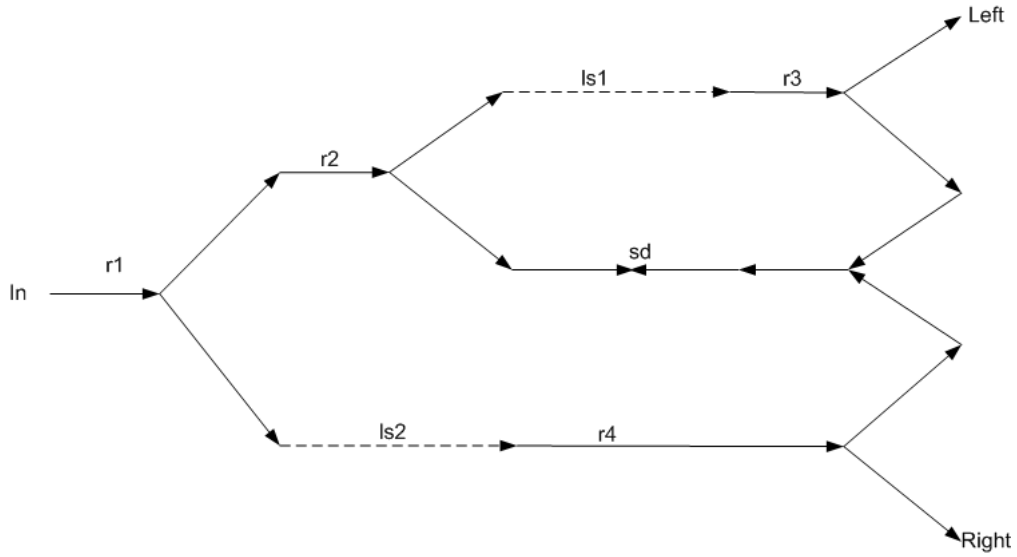

Figure 5.3 – Connecteur XOR-Split en Reo

Implémentation Java

Ce connecteur est déjà implémenté dans Reo et se nomme *ExclusiveRouter*. Notre connecteur a une entrée (*in*) et deux sorties (*left* et *right*). Voici le corps de notre classe Java.

```
ExclusiveRouter er = new ExclusiveRouter();
in = er.getIn();
left = er.getLeft();
right = er.getRight();
```

5.1.4 XOR-Join

Le connecteur *XOR-Join* permet à deux canaux de se regrouper en un canal mais seul le flux d'un des deux canaux en entrée transitera vers la canal de sortie.

Modélisation Reo

La figure 5.4 représente le connecteur *XOR-Join* en Reo.

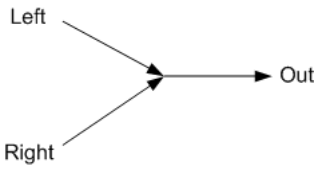

Figure 5.4 – Connecteur XOR-Join en Reo

Implémentation Java

Ce connecteur est déjà implémenté dans Reo et se nomme *Merger*. Notre connecteur a deux entrées (*left* et *right*) et une sortie (*out*). Voici le corps de notre classe Java.

```
Merger merger = new Merger();
left = merger.getLeft();
right = merger.getLeft();
out = merger.getOut();
```

5.1.5 OR-Split

Le connecteur *OR-Split* permet à un canal de se diviser en deux canaux et le flux transite dans un des deux canaux ou dans les deux.

Modélisation Reo

La figure 5.5 représente le connecteur *OR-Split* en Reo.

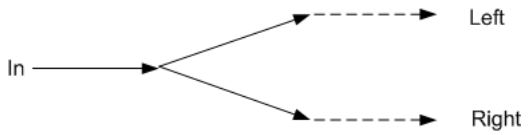

Figure 5.5 – Connecteur OR-Split en Reo

Implémentation Java

Nous utilisons le connecteur *Replicator* et deux connecteurs *LossySync* qui sont déjà implémentés dans Reo. Notre connecteur a une entrée (*in*) et deux sorties (*left* et *right*). Voici le corps de notre classe Java.

```
Replicator r = new Replicator();
LossySync ls1 = new LossySync();
LossySync ls2 = new LossySync();
r.getLeft().JOIN(ls1.getLeft());
r.getRight().JOIN(ls2.getLeft());
in = r.getIn();
left = ls1.getRight();
right = ls2.getRight();
```

5.1.6 OR-Join

Le connecteur *OR-Join* permet à deux canaux de se regrouper en un canal et le flux d'un ou des deux canaux en entrée transitera vers le canal de sortie.

Nous n'allons pas traiter ce connecteur ici mais dans une section suivante (*Control Workflow Patterns*) car il y a 3 possibilités d'implémentations qui correspondent aux pattern 7 (*Synchronizing merge*), pattern 8 (*Multi-Merge*) et pattern 9 (*Discriminator*).

5.2 Tâches

Le système choisi implémente chaque tâche sous forme d'un *thread* Java. Le *thread* attend la réception d'un message pour s'exécuter. Lorsqu'il reçoit ce message, il peut débuter sa tâche. À la fin de son exécution, il envoie un message aux tâches suivantes. Voici l'implémentation du corps de la méthode *run* qui est lancée quand on appelle la méthode *start* d'un *thread*.

```
OutputEnd oe = new OutputEnd();
InputEnd ie = new OutputEnd();
...
oe.connect(this);
Message c1 = (Message) oe.take();
// Traitement
ie.connect(this);
ie.write(new Message(nom, nom));
```

Dans notre exemple le message contient le nom de la tâche. Nous examinerons les messages plus en profondeur avec la perspective de données.

Une tâche atomique a un comportement relativement basique. Elle s'exécute et puis passe le contrôle à la tâche suivante.

Une tâche composée est divisée en quatre parties :
- *l'input condition* est le point d'entrée de la tâche. C'est elle qui envoie le premier message dans la tâche composée ;
- la ou les tâche(s) de la tâche composée qui sont stockées dans un tableau ;
- le ou les état(s) (*condition*) de la tâche composée qui sont également stockés dans un tableau ;
- *l'output condition* est le point de sortie de la tâche. C'est elle qui va recevoir le dernier message de la tâche composée.

La classe *TaskComposite* a une méthode *startWorkflow* qui est appelée par la tâche composée *root* ou dans l'autre cas par la méthode *run* d'une tâche composée. Cette méthode permet de lancer le *thread* pour chaque tâche qui compose la tâche composée.

Étant donné que nous avons des caractéristiques communes entre *TaskAtomic*, *TaskComposite* et *Condition*, nous créons une classe abstraite *ObjectWorkflowAbstract* afin de ne définir qu'une seule fois certains attributs et certaines méthodes nécessaires dans les deux cas. Il y a également une classe *TaskAbstract* qui hérite de *ObjectWorkflowAbstract* pour les attributs qui correspondent seulement à une tâche.

Le code complet de ces classes est disponible en annexe C.2.

5.3 Instance de tâches

Tout comme les tâches nous avons une classe abstraite *TaskInstanceAbstract*. Cette classe va gérer les différentes instances d'une tâche. Elle contient un tableau d'*Instance* qui représente l'instance elle-même. Cette classe est également une classe abstraite.

La différence entre une tâche atomique et une tâche composée est la création de l'instance. La tâche atomique va instancier la classe *InstanceAtomic* et la tâche composée va instancier la classe *InstanceComposite*.

Une instance de tâche atomique a un comportement relativement basique. Elle s'exécute et puis passe le contrôle à la tâche suivante en appelant la méthode *writer* de *TaskInstanceAbstract* afin de gérer la synchronisation des instances si nécessaire.

Une instance de tâche composée va appeler la méthode *startWorkflow* de sa tâche composée. Après son exécution elle va également appeler la méthode *writer* de *TaskInstanceAbstract* afin de gérer la synchronisation des instances si nécessaire.

Le code complet de ces classes est disponible en annexe C.2.

5.4 Control Workflow Patterns

Dans cette section, nous vérifions pour chaque pattern de contrôle de flux s'il est possible de les implémenter avec le langage Reo. Ces patterns ont été élaborés par [van der Aalst *et al.*, 2003].

Nous présentons dans cette partie l'explication de l'implémentation du pattern à l'aide du code Java.

5.4.1 Basic control-flow patterns

Ces patterns représentent la définition des concepts élémentaires de contrôle de flux fournis par la WfMC.

Pattern 1 : Sequence

Le pattern *Sequence* permet à une tâche du processus de workflow de commencer quand la tâche qui la précède dans le même processus est finie.

Implémentation

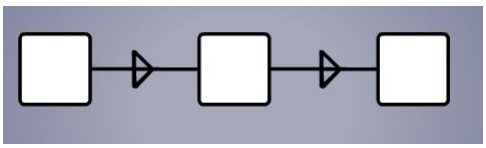

Figure 5.6 – Control Workflow Patterns - Sequence

La figure 5.6 représente un exemple du pattern *Sequence*. Nous avons 3 tâches atomiques que nous appellerons A, B et C et deux canaux qui relient la tâche A à B et la tâche B à C. Voici le code Java nécessaire pour exécuter ce workflow.

```
SyncChannel channelAToB = new SyncChannel();
SyncChannel channelBToC = new SyncChannel();
TaskComposite t = new TaskComposite();
t.addTask(new TaskAtomic(null, channelAToB.getLeft(), "A"));
t.addTask(new TaskAtomic(channelAToB.getRight(), channelBToC.getLeft(), "B"));
t.addTask(new TaskAtomic(channelBToC.getRight(), null, "C"));
t.startWorkflow();
```

Notre workflow est représenté par une tâche composée qui contient 3 tâches. Une fois l'initialisation finie, nous lançons le workflow avec la méthode *startWorkflow*.

Imaginons maintenant que la tâche B est également une tâche composée qui contient 3 tâches atomiques que nous appellerons B1, B2 et B3. Nous aurons besoin de deux canaux supplémentaires pour relier B1 à B2 et B2 à B3.

```
SyncChannel channelB1ToB2 = new SyncChannel();
SyncChannel channelB2ToB3 = new SyncChannel();
```

Voici la déclaration de la tâche B que nous avons appelé $t2$ et puis nous l'ajoutons à notre tâche composée qui représente notre workflow.

```
TaskComposite t2 = new TaskComposite(channelAToB.getRight(),
                                     channelBToC.getLeft(), "B");
t2.addTask(new TaskAtomic(null, channelB1ToB2.getLeft(), "B1"));
t2.addTask(new TaskAtomic(channelB1ToB2.getRight(), channelB2ToB3.getLeft(),
                          "B2"));
t2.addTask(new TaskAtomic(channelB2ToB3.getRight(), null, "B3"));
t.addTask(t2);
```

Pattern 2 : Parallel Split

Le pattern *Parallel Split* permet de diviser une branche en plusieurs branches parallèles. Les tâches sur chaque branche s'exécutent simultanément.

Implémentation

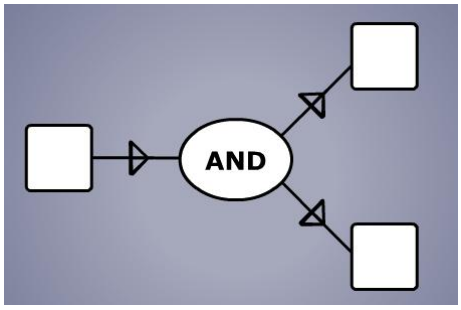

Figure 5.7 – Control Workflow Patterns - Parallel Split

La figure 5.7 représente un exemple du pattern *Parallel Split*. Nous avons 3 tâches atomiques que nous appellerons A, B et C et un connecteur AND-Split qui empruntera un des chemins au hasard. Voici le code Java nécessaire pour exécuter ce workflow.

```
AndSplit andSplit = new AndSplit();
TaskComposite t = new TaskComposite();
t.addTask(new TaskAtomic(null, andSplit.getIn(), "A"));
t.addTask(new TaskAtomic(andSplit.getLeft(),null, "B"));
t.addTask(new TaskAtomic(andSplit.getRight(), null, "C"));
t.startWorkflow();
```

Pattern 3 : Synchronization

Le pattern *Synchronization* permet de faire converger, en un seul point de synchronisation, de multiples tâches parallèles.

Implémentation

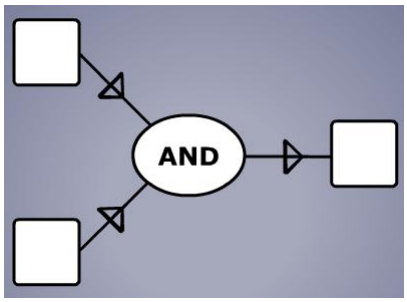

Figure 5.8 – Control Workflow Patterns - Synchronization

La figure 5.8 représente un exemple du pattern *Synchronization*. Nous avons 3 tâches atomiques que nous appellerons A, B et C et un connecteur AND-Join. Voici le code Java dont nous avons besoin pour exécuter ce workflow.

```
AndJoin andJoin = new AndJoin();
TaskComposite t = new TaskComposite();
t.addTask(new TaskAtomic(null, andJoin.getLeft(), "A"));
t.addTask(new TaskAtomic(null, andJoin.getRight(), "B"));
t.addTask(new TaskAtomic(andJoin.getOut(), null, "C"));
t.startWorkflow();
```

Pattern 4 : Exclusive Choice

Le pattern *Exclusive Choice* permet de choisir, sur base de paramètres de décision ou de workflow, une branche parmi plusieurs.

Implémentation

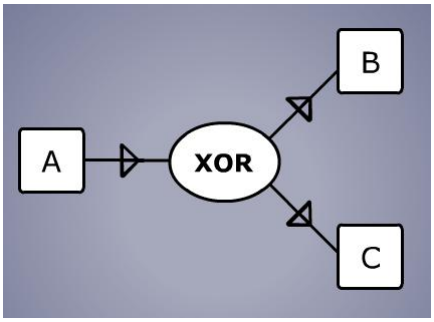

Figure 5.9 – Control Workflow Patterns - Exclusive Choice

La figure 5.9 représente un exemple du pattern *Exclusive Choice*. Nous avons 3 tâches atomiques que nous appellerons A, B et C et un connecteur XOR-Split qui empruntera un des deux chemins au hasard. Voici le code Java dont nous avons besoin pour exécuter ce workflow.

```
XorSplit xorSplit = new XorSplit();
TaskComposite t = new TaskComposite();
t.addTask(new TaskAtomic(null, xorSplit.getIn(), "A"));
t.addTask(new TaskAtomic(xorSplit.getLeft(),null, "B"));
t.addTask(new TaskAtomic(xorSplit.getRight(), null, "C"));
t.startWorkflow();
```

Pattern 5 : Simple Merge

Le pattern *Simple Merge* permet de faire converger, sans synchronisation, de multiples tâches parallèles. Il est implicite dans ce pattern qu'aucune des branches alternatives n'est exécutée en parallèle (si ce n'est pas le cas, voir alors le pattern 8 (*Multi-merge*) ou le pattern 9 (*Discriminator*)).

Implémentation

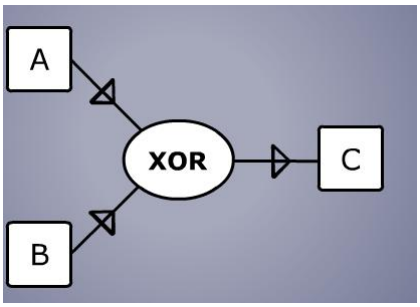

Figure 5.10 – Control Workflow Patterns - Simple Merge

La figure 5.10 représente un exemple du pattern *Simple Merge*. Nous avons 3 tâches atomiques que nous appellerons A, B et C et un connecteur XOR-Join. Voici le code Java dont nous avons besoin pour exécuter ce workflow.

```
XorJoin xorJoin = new XorJoin();
TaskComposite t = new TaskComposite();
t.addTask(new TaskAtomic(null, xorJoin.getLeft(), "A"));
t.addTask(new TaskAtomic(null, xorJoin.getRight(), "B"));
t.addTask(new TaskAtomic(xorJoin.getOut(), null, "C"));
t.startWorkflow();
```

5.4.2 Advanced Branching and Synchronization Patterns

Dans cette section, nous mettons en avant des modèles plus avancés de branchement et de synchronisation.

Pattern 6 : Multi-choice

Le pattern *Multi-choice* permet de choisir, sur base de paramètres de décision ou de workflow, un certain nombre de branches.

Implémentation

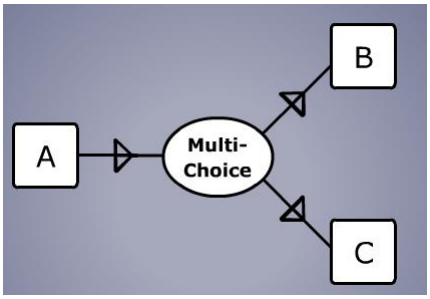

Figure 5.11 – Control Workflow Patterns - Multi-choice

La figure 5.11 représente un exemple du pattern *Multi-Choice*. Nous avons 3 tâches atomiques que nous appellerons A, B et C et un connecteur OR-Split. Nous choisissons la branche C dans notre workflow d'exemple. Pour cela nous devons associer une condition XPath à chaque branche. La condition $false()$ va être associée à la branche de A vers B et la condition $true()$ à la branche de A vers C.

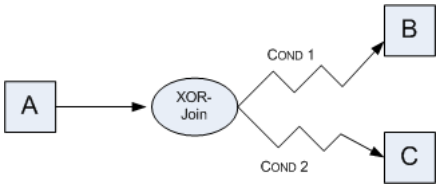

Figure 5.12 – Control Workflow Patterns - Multi-choice en Reo

Nous illustrons cette implémentation à l'aide de la figure 5.12 qui est son implémentation en Reo. On remarque qu'on utilise le canal *Filter* sur chaque branche afin de pouvoir associer notre condition XPath.

La version de Reo que nous avons choisi possède malheureusement quelques limitations. L'une de celles-ci est l'implémentation du canal *Filter*. En effet, on ne peut pas forcer le choix entre deux alternatives. Par exemple, lors d'un OR-Split ou du XOR-Split, Reo empruntera un des deux chemins au hasard. Il n'y a donc pas moyen de le forcer à suivre un chemin défini lors de la conception du workflow. Dans notre implémentation, nous avons décidé de pallier à ce défaut. Voici le code Java dont nous avons besoin pour exécuter ce workflow avec notre version ReoLite.

```
OrSplit orSplit = new OrSplit();
TaskComposite t = new TaskComposite();
TaskAtomic a = new TaskAtomic(null, orSplit.getIn(), "A");
a.addFlows("B", "false()");
a.addFlows("C", "true()");
t.addTask(a);
t.addTask(new TaskAtomic(orSplit.getRight(), null, "B", a.getPredicate("B")));
t.addTask(new TaskAtomic(orSplit.getLeft(), null, "C", a.getPredicate("C")));
t.startWorkflow();
```

Nous encodons au niveau de A les différents flux et les conditions qui sont possibles grâce à la méthode *addFlows* qui prend deux paramètres. Le premier est l'identifiant de la prochaine tâche et le second est la condition en XPath.

Pattern 7 : Synchronizing merge

Le pattern *Synchronizing merge* permet de faire converger de multiples tâches parallèles. Si plusieurs chemins sont empruntés, alors il y a synchronisation avant de commencer la tâche suivante. Il est implicite dans ce pattern qu'une branche qui a déjà été activée, ne peut plus être réactivée tant qu'on attend toujours d'autres branches pour terminer la synchronisation.

Implémentation

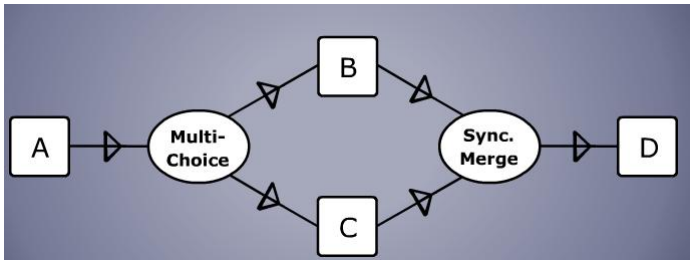

Figure 5.13 – Control Workflow Patterns - Synchronizing merge

La figure 5.13 représente un exemple du pattern *Synchronizing merge*. Nous avons 4 tâches atomiques que nous appellerons A, B, C et D. Nous utilisons le pattern que nous venons de voir et qui est *Multi-choice* pour effectuer le choix.

La tâche D ne peut être exécutée qu'une fois que la tâche B ou C ou les deux sont finies en fonction du choix effectué. Notre solution pour ce pattern est d'utiliser le connecteur *AND-Split* et d'envoyer un message avec un format reconnaissable par le système lorsqu'une tâche ne doit pas être exécutée. Ce système permet à notre connecteur de toujours recevoir autant de jetons qu'il y a de tâches. Ce message sera ignoré par les autres tâches.

Pattern 8 : Multi-merge

Le pattern *Multi-merge* permet de faire converger de multiples tâches parallèles sans synchronisation. Si plus d'une branche est activée, probablement de façon concomitante, la tâche suivant la fusion est commencée à chaque branche entrante.

Implémentation

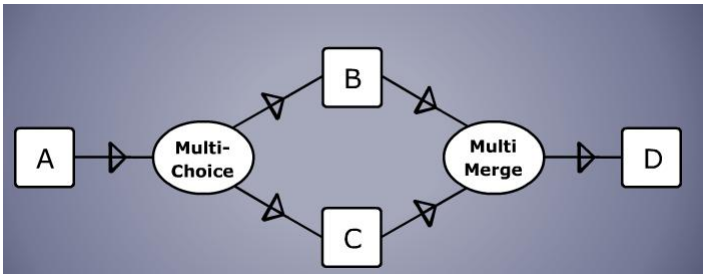

Figure 5.14 – Control Workflow Patterns - Multi-merge

La figure 5.14 représente un exemple du pattern *Multi-merge*. Nous devrons exécuter D une fois si on choisit B ou C mais deux fois si on choisit les deux. Notre solution pour ce pattern est de dédoubler D. Nous aurons une tâche D1 après B et une tâche D2 après C. D1 et D2 sont identiques, c'est à dire, qu'elles exécutent toutes les deux la tâche D. Nous aurons un connecteur

AND-Join avec en entrée D1 et D2 et en sortie la suite du workflow que nous n'avons pas dans ce cas-ci.

Pattern 9 : Discriminator

Le pattern *Discriminator* permet de faire converger de multiples tâches parallèles. Il attend une des branches entrantes pour accomplir la tâche suivante. À ce moment toutes les branches restantes à accomplir sont ignorées. Une fois que toutes les branches entrantes ont été déclenchées, il est réinitialisé de sorte qu'il puisse être déclenché à nouveau.

Implémentation

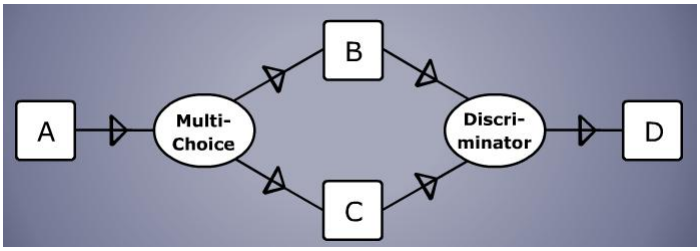

Figure 5.15 – Control Workflow Patterns - Discriminator

La figure 5.15 représente un exemple du pattern *Discriminator*. Celui-ci est le même pattern que le pattern *Synchronizing merge* mais ici le connecteur *Discriminator* va être implémenté avec un connecteur *XOR-Join*. Ce connecteur permet d'ignorer tous les autres jetons qui pourraient arriver après.

5.4.3 Structural patterns

Certains systèmes de workflow imposent des restrictions d'un point de vue structurel sur tous les workflow pouvant être construits. Ce type de restriction est exprimé à travers les patterns suivants.

Pattern 10 : Arbitrary Cycles

Le pattern *Arbitrary Cycles* permet à une ou plusieurs tâches d'être effectuées à plusieurs reprises.

Implémentation

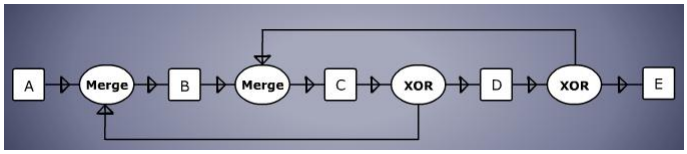

Figure 5.16 – Control Workflow Patterns - Arbitrary Cycles

La figure 5.16 représente un exemple du pattern « *Arbitrary Cycles* ». Nous utilisons le connecteur *Xor-Join* pour implémenter le *Merge* et nous utilisons le connecteur *Xor-Split* pour implémenter le *XOR*.

Pattern 11 : Implicit terminaison

Le pattern *Implicit terminaison* permet de terminer le workflow quand il n'y a plus de tâche active. En d'autres termes, il n'y a aucune tâche active dans le workflow et aucune autre tâche qui pourrait devenir active. De plus, le workflow n'est pas en *deadlock*.

Implémentation

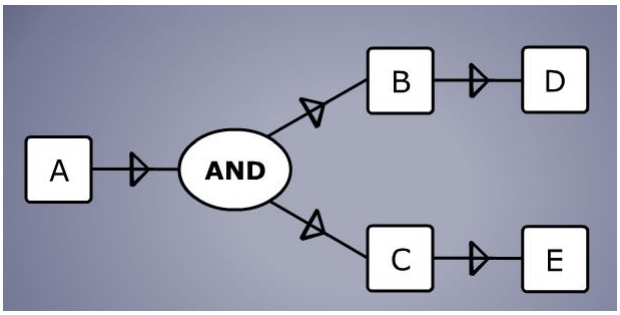

Figure 5.17 – Control Workflow Patterns - Implicit terminaison

La figure 5.17 représente un exemple du pattern de *Implicit terminaison*. Nous allons après la tâche D et E rajouter une tâche F. F ne peut être atteinte qu'une fois qu'on a fini D et E. Nous utilisons le connecteur AND-Join avec en entrée D et E et en sortie F.

5.4.4 Patterns involving multiple instances

Dans cette section, nous présentons les patterns qui se réfèrent à de multiples instances.

Pattern 12 : Multiple Instances Without Synchronization

Le pattern *Multiple Instances Without Synchronization* permet de créer plusieurs instances d'une tâche. Chacune de ces instances peut être encapsulée dans une *thread*. Néanmoins, il n'y a aucun besoin de synchroniser ces threads.

Implémentation

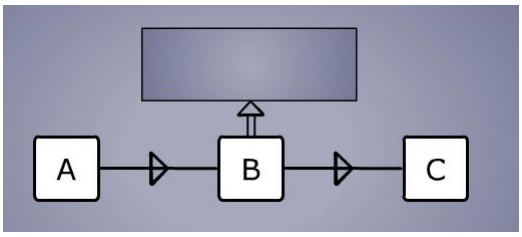

Figure 5.18 – Control Workflow Patterns - Multiple Instances Without Synchronization

La figure 5.18 représente un exemple du pattern Multiple Instances. Nous avons 3 tâches atomiques que nous appellerons A, B et C. La tâche B permet d'avoir de multiples instances de cette tâche atomique. Voici le code Java dont nous avons besoin pour exécuter ce workflow.

```
SyncChannel channelAToB = new SyncChannel();
SyncChannel channelBToC = new SyncChannel();
TaskComposite t = new TaskComposite();
t.addTask(new TaskAtomic(null, channelAToB.getLeft(), "A"));
TaskInstanceAtomic b = new TaskInstanceAtomic(channelAToB.getRight(),
                                              channelBToC.getLeft(), "B");
b.setNbreInstanceDepart(0);
b.ajoutDynamique();
b.setSynchronization(false);
t.addTask(b);
t.addTask(new TaskAtomic(channelBToC.getRight(), null, "C"));
t.startWorkflow();
```

Nous remarquons que maintenant la tâche B appelle le constructeur *TaskInstanceAtomic* qui va permettre de créer une classe qui va gérer les instances multiples de tâche atomique.

Dans ce cas, nous n'avons pas d'instance à créer au départ, donc nous allons initialiser le nombre d'instances de départ à 0 (*setNbreInstanceDepart*). Nous indiquons également à la classe que nous ne souhaitons pas de synchronisation des différentes instances (*setSynchronization(false)*). Nous pourrons donc exécuter C avant que les différentes instances soient finies. Nous indiquons que nous souhaitons ajouter des instances en cours d'exécution en appelant la méthode (*ajoutDynamique*).

Pattern 13 : Multiple Instances With a Priori Design Time Knowledge

Le pattern *Multiple Instances With a Priori Design Time Knowledge* permet de créer plusieurs instances d'une tâche. Le nombre d'instances de la tâche est connu lors de la conception. Une fois que toutes les instances sont finies alors la tâche suivante peut commencer.

Implémentation

Voici l'initialisation de B pour ce cas-ci.

```
TaskInstanceAtomic b = new TaskInstanceAtomic(channelAToB.getRight(),
                                              channelBToC.getLeft(), "B");
b.setNbreInstanceDepart(3);
t.addTask(b);
```

Nous avons fixé au design du workflow que nous ne voulions que 3 instances. Ici, nous ne précisions pas si nous voulons ou pas une synchronisation. Le système est en mode synchrone par défaut.

Pattern 14 : Multiple Instances With a Priori Runtime Knowledge

Le pattern *Multiple Instances With a Priori Runtime Knowledge* est similaire au précédent. La différence est que le nombre d'instances est connu lors de l'exécution plutôt que lors de la conception. Toutefois, le nombre d'instances doit être connu avant l'exécution de cette tâche.

Implémentation

Voici l'initialisation de B pour ce cas-ci.

```
TaskInstanceAtomic b = new TaskInstanceAtomic(channelAToB.getRight(),
                                              channelBToC.getLeft(), "B");
t.addTask(b);
```

Si nous ne précisons pas le nombre d'instances au départ, le système va le demander lors de l'exécution.

Pattern 15 : Multiple Instances With No a Priori Runtime Knowledge

Le pattern *Multiple Instances With No a Priori Runtime Knowledge* est une généralisation du pattern 14 (*Multiple Instances With a Priori Runtime Knowledge*). Ici, même lors de l'exécution d'une tâche, de nouvelles instances de cette tâche peuvent être ajoutées. Donc, il est toujours possible de rajouter une instance à une tâche.

Implémentation

Voici l'initialisation de B pour ce cas-ci.

```
TaskInstanceAtomic b = new TaskInstanceAtomic(channelAToB.getRight(),
                                              channelBToC.getLeft(), "B");
b.ajoutDynamique();
t.addTask(b);
```

Le principe est identique au pattern précédent mais ici on indique qu'on souhaite ajouter des instances en cours d'exécution en appelant la méthode (« *ajoutDynamique* »).

5.4.5 State-based patterns

Le point commun des patterns suivants est qu'ils ont tous un rapport avec l'état en cours du workflow.

Pattern 16 : Deferred Choice

Le pattern *Deferred Choice* permet de choisir une branche parmi plusieurs. Contrairement au XOR-Split, le choix n'est pas fait explicitement (par exemple basé sur des données ou sur une décision) mais plusieurs solutions alternatives sont offertes à l'environnement. Cependant, contrairement au AND-Split, seulement une des solutions alternatives est exécutée. Ceci signifie qu'une fois que l'environnement active une des branches, les autres branches alternatives sont retirées. Il est important de noter que le choix est retardé jusqu'à ce que le traitement dans une des branches alternatives soit réellement commencé. C'est-à-dire que le moment du choix est aussi tardif que possible.

Implémentation

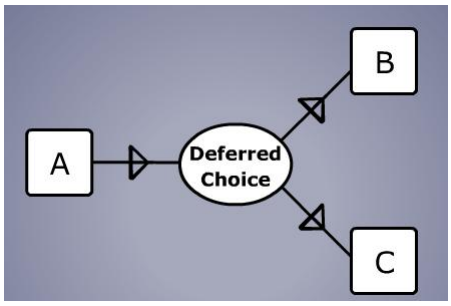

Figure 5.19 – Control Workflow Patterns - Deferred Choice

La figure 5.19 représente un exemple du pattern *Deferred Choice*. Ce pattern est similaire au pattern « *Multi-choice* ». Nous allons indiquer une condition sur les branches de A à B et de B à C. La différence ici est que la condition XPath va examiner la valeur d'une variable encodée par l'utilisateur dans une tâche précédente. Cette valeur est évaluée au moment du choix de la branche.

Pattern 17 : Interleaved Parallel Routing

Le pattern *Interleaved Parallel Routing* permet à un ensemble de tâches d'être exécutées dans un ordre arbitraire. Chaque tâche est exécutée à son tour et l'ordre est décidé à l'exécution.

Implémentation

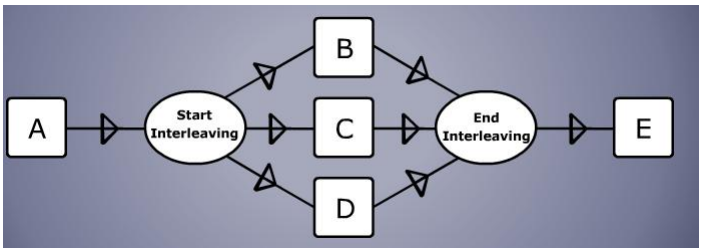

Figure 5.20 – Control Workflow Patterns - Interleaved Parallel Routing

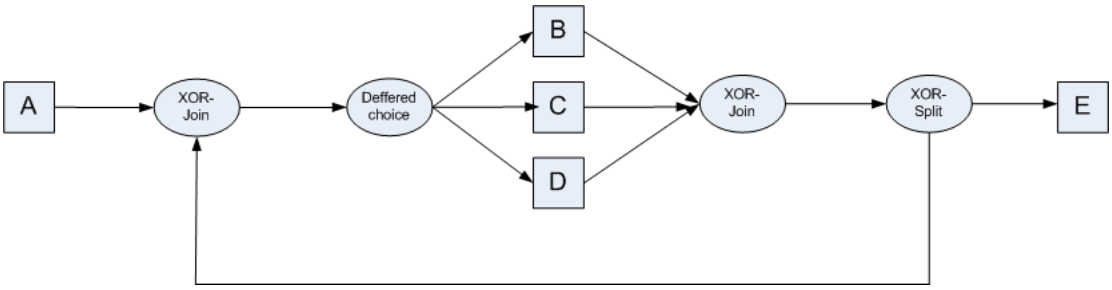

Figure 5.21 – Control Workflow Patterns - Interleaved Parallel Routing en Reo

La figure 5.20 représente un exemple du pattern *Interleaved Parallel Routing*. Nous illustrons cette implémentation à l'aide de la figure 5.21. Nous effectuons une boucle autour de la tâche B, C et D et à chaque itération, nous utilisons le pattern *Deferred choice* afin de dire quel est notre

choix au cours de l'exécution du workflow. Au premier tour, on choisit la première tâche qu'on veut exécuter, au deuxième tour on choisit la deuxième tâche et au dernier tour, on choisit la dernière. Nous utilisons le connecteur *XOR-Join* et le connecteur *XOR-Split* pour effectuer la boucle.

Pattern 18 : Milestone

Le pattern *Milestone* permet d'activer une tâche si une autre tâche est dans un certain état. C'est-à-dire que la tâche n'est seulement activée que si on a atteint une certaine étape importante qui n'a pas encore expirée.

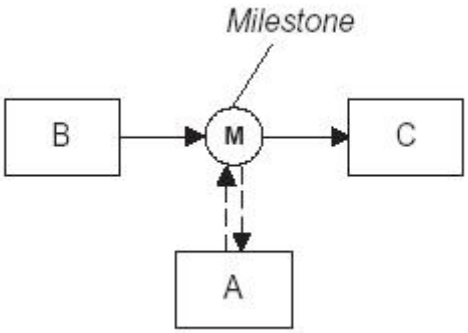

Figure 5.22 – Représentation schématique du milestone [van der Aalst *et al.*, 2003]

La figure 5.22 montre un exemple concret avec trois tâches appelées A, B, C. La tâche A est seulement permise si la tâche B a été exécutée et C n'a pas encore été exécutée, c'est-à-dire A n'est pas permise avant l'exécution de B et A n'est pas permise après l'exécution de C. L'état entre B et C est représenté par M. M est une étape importante pour A. Notons que A n'enlève pas le jeton de M, il examine seulement la présence du jeton.

Implémentation

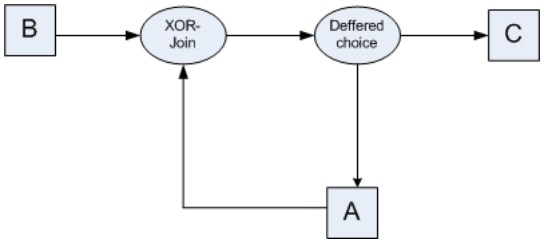

Figure 5.23 – Control Workflow Patterns - Milestone en Reo

La figure 5.23 illustre l'implémentation en Reo du pattern *Milestone*. Nous utilisons le pattern *Deferred choice* afin de dire quel est notre choix au cours de l'exécution du workflow. Nous effectuons une boucle autour de la tâche A à l'aide du connecteur *XOR-Join*.

5.4.6 Cancellation patterns

Les patterns suivants concernent l'annulation d'une tâche ou d'une instance de workflow.

Pattern 19 : Cancel activity

Le pattern *Cancel activity* permet à une tâche active de se rendre inactive.

Implémentation

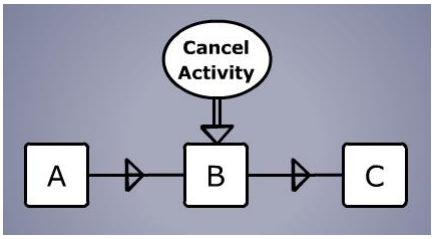

Figure 5.24 – Control Workflow Patterns - Cancel activity

La figure 5.24 représente un exemple du pattern *Cancel activity*. Nous avons 3 tâches atomiques dont une qu'on peut annuler pendant son exécution. Une fois qu'on a reçu un événement Java qui annonce qu'on annule la tâche, on envoie un message d'annulation de la tâche (*CANCELTASK*) à la ou les tâche(s) suivante(s).

Pattern 20 : Cancel case

Le pattern *Cancel case* permet d'annuler un cas, c'est-à-dire une instance de workflow.

Implémentation

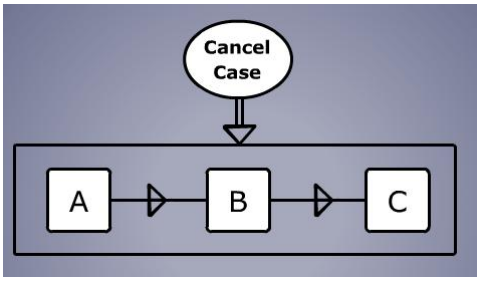

Figure 5.25 – Control Workflow Patterns - Cancel case

La figure 5.25 représente un exemple du pattern *Cancel case*. Nous avons 3 tâches atomiques et à tout moment, nous pouvons recevoir un événement Java pour annuler le workflow. Si on reçoit cet événement, on envoie un message d'annulation du workflow (*CANCELWORKFLOW*) à chaque tâche qui est en cours d'exécution. Chaque tâche suivante qui va recevoir le message ne va pas s'exécuter et va le renvoyer à sa ou ses tâches suivante(s) afin d'atteindre la fin du workflow.

5.5 Conclusion

Nous venons d'expliquer l'implémentation, à l'aide de Reo et de Java, de la perspective de contrôle de flux. Nous avons vérifié que notre implémentation pouvait être utilisée pour chaque pattern de contrôle de flux [van der Aalst *et al.*, 2003].

Pendant la rédaction de ce mémoire, une nouvelle version de ces patterns a été publiée [Russell *et al.*, 2006]. Le contenu de ce chapitre est toujours correct mais pas complet étant donné que de nouveaux patterns ont été rajoutés à la version précédente.

Dans le chapitre suivant, nous allons analyser et implémenter la perspective données.

Chapitre 6

Perspective de données

Ce chapitre présente les workflow sous la perspective des données. À l'instar de la perspective de contrôle de flux, nous présentons cette perspective à travers le fichier XML que l'on reçoit puis aux travers des différents patterns identifiés dans la littérature. Pour implémenter les patterns nous devons choisir notre type d'implémentation. Nous avons deux choix qui s'offrent à nous. Le premier est la manipulation de fichier XML tandis que le second est la manipulation des données par un langage de coordination. Nous présentons les avantages et les inconvénients de chacun. Puis nous effectuons notre choix d'implémentation afin de pouvoir présenter les patterns de données.

6.1 Parsing du fichier XML

Dans cette section nous analysons la structure du fichier XML pour les données de notre workflow.

La figure 6.1 représente la structure d'une déclaration de variable. Le tag *complexType* possède un attribut *name* qui permet de donner un nom au nouveau type de variable. Le tag *sequence* et *element* ont deux attributs communs qui sont *minOccurs* et *maxOccurs*. Ceux-ci représentent le nombre minimum et maximum de fois qu'on va retrouver cette variable. Dans le tag *element* on a deux variantes possibles : soit la déclaration d'un élément avec comme attribut un nom et le type de l'élément, soit on retrouve la déclaration d'un type complexe.

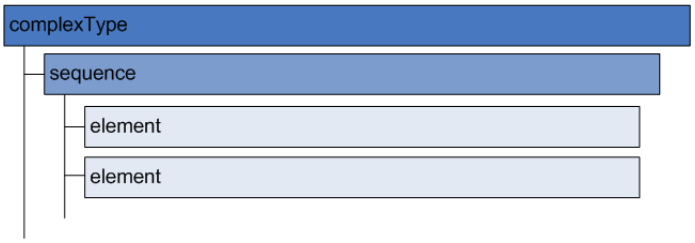

Figure 6.1 – Structure du fichier XML - Déclaration de variable

La figure 6.2 représente la structure pour déclarer une variable locale dans une tâche. Le tag *decomposition* est le même que celui expliqué dans la perspective précédente. Nous avons complété la figure 4.1 afin d'intégrer les données. Pour chaque variable on déclare le nom, le type, une valeur initiale et le *namespace* qui est le chemin vers le schéma XML.

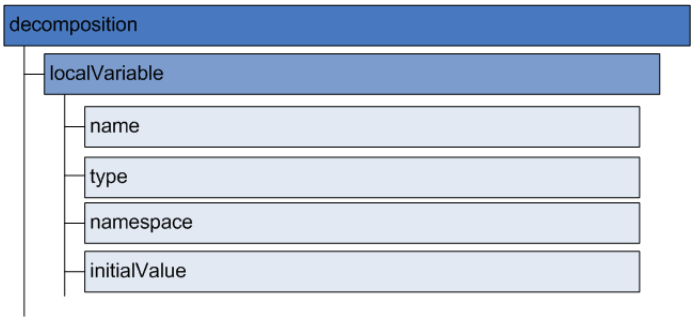

Figure 6.2 – Structure du fichier XML - Variable d'une tâche

La figure 6.3 représente la structure pour déclarer des paramètres d'entrées (*inputParam*) et des paramètres de sorties (*outputParam*) à une tâche. Ces données se trouvent également dans le tag *decomposition* que nous avons expliqué à l'aide de la figure 4.1. Pour chaque paramètre on déclare le nom, le type et le *namespace*.

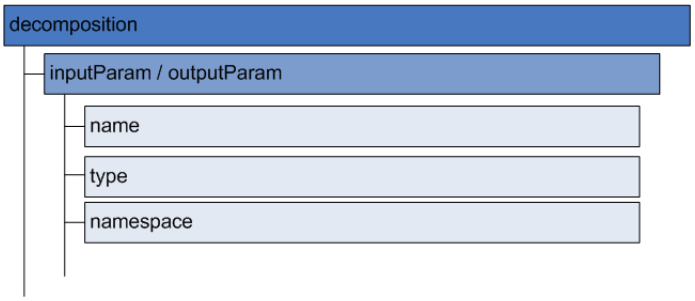

Figure 6.3 – Structure du fichier XML - Paramètre d'une tâche

La figure 6.4 représente la structure pour effectuer le mapping entre les paramètres sortants d'une tâche et les paramètres entrants. Ces données viennent compléter la figure 4.1 de la perspective prédédente. *startingMappings* effectue le mapping pour les paramètres entrants et *completeMappings* effectue le mapping pour les paramètres sortants. On a autant de tags *mapping* qu'il y a de paramètres entrants ou sortants. Pour chacun on a une *expression* XQuery qui indique où trouver la valeur de la variable. On a également le nom de la variable dans la tâche qui va recevoir cette valeur (*mapTo*).

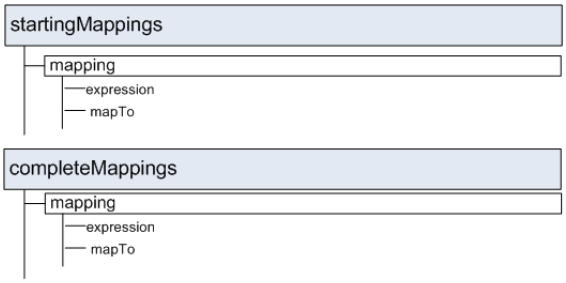

Figure 6.4 – Structure du fichier XML - Mapping

6.2 Implémentation

Nous distinguons deux possibilités pour implémenter cette perspective. La première consiste à manipuler les données dans des fichiers XML et la deuxième consiste à manipuler les données dans des espaces de tuples. Nous exposons ces deux implémentations, puis nous en choisissons une que nous utiliserons dans la suite de ce chapitre.

6.2.1 Fichier XML

Dans le fichier de modélisation du workflow, XPath et XQuery sont utilisés afin de décrire la manipulation des fichiers XML au sein du moteur de workflow.

Il y a donc un fichier XML par instance de tâches qui contient les données de la tâche. Pour effectuer le passage d'informations entre deux tâches, on passe par la tâche composée de ces tâches. Supposons que A est une tâche composée de B et de C alors :
- les données sont stockées au niveau de la tâche composée en `localVariabe` ;
- les sous-tâches de la tâche composée déclarent les paramètres d'entrées et de sorties.

Au début de la tâche, on effectue un mapping entre les variables de la tâche composée et les paramètres d'entrées de la tâche (*inputParam*). Tandis qu'à la sortie de la tâche, on effectue un mapping des paramètres de sorties (*outputParam*) de la tâche vers les variables de la tâche composée. Les mapping sont effectués grâce à XPath.

Le moteur de workflow affiche les paramètres d'entrées en lecture seule car ce sont des informations reçues des tâches précédentes. Le moteur de workflow affiche les paramètres de sorties en écriture car ce sont des informations à encoder dans la tâche courante.

Pour rappel, dans le cas d'un OR-Split ou d'un XOR-Split, il est possible d'ajouter une condition sur la branche qui est encodée dans le fichier de modélisation (à l'aide de XPath). Celle-ci sera évaluée au moment de l'exécution.

Notons que pour les instances multiples, il faut diviser les données de chaque instance. À la terminaison de toutes les instances, on doit regrouper les différentes données des instances pour les passer à la tâche suivante. Pour ce faire, on utilise XQuery.

6.2.2 Espace de tuples

Une autre solution est de stocker les données dans un espace de tuples. Cette implémentation est effectuée grâce au langage de coordination KLAVA.

Chaque tâche ou instance de tâche va avoir un espace de tuples (*TupleSpace*) qui lui est propre afin de pouvoir stocker ses variables localement. Il y a aura un *Tuple* par variable. Il contiendra le nom et le contenu de la variable. Chaque tâche ou instance de tâches a une référence vers l'espace de tuples de son parent. Quand la tâche voudra utiliser une variable, elle regardera d'abord dans son espace de tuples et s'il n'y est pas alors elle regardera dans celui de son parent et ainsi de suite jusqu'à trouver la variable dont elle a besoin.

Au final, on construit donc un arbre d'espace de tuples, dans la mesure où chaque sous-tâche peut accéder à l'espace de tuples de son parent et ainsi de suite jusqu'à atteindre l'espace de tuples de la tâche composée *root*.

Notre première idée était de partir du fichier de modélisation en utilisant les données XPath et XQuery. Si nous choisissons cette solution, nous sommes obligés d'établir un dictionnaire

avec tous les *mapping* possibles pour les entrées et sorties de la tâche. Nous sommes également confrontés au fait que nous devons interpréter les requêtes XQuery afin de pouvoir les appliquer à nos espaces de tuples.

Etant donnée la complexité engendrée, la meilleure solution est d'effectuer un *refactoring* de la structure du fichier de modélisation en supprimant la gestion des données par XPath et XQuery. Il faudra donc changer le schéma XML de notre fichier de modélisation afin de pouvoir encoder les données relatives à la gestion de données dans l'idée d'un arbre d'espace de tuples.

6.2.3 Choix

Qu'il s'agisse d'un fichier XML ou d'un espace de tuples, au final nous n'avons toujours qu'une structure de données. Vu les changements majeurs à effectuer pour pouvoir utiliser la solution qui utilise un langage de coordination, nous avons choisi d'implémenter cette perspective à l'aide de fichiers XML. Ce choix d'implémentation est celui qui demande le moins d'effort d'implémentation au sein du moteur, vu que toute l'information est déjà présente dans le fichier de modélisation.

6.3 Data Workflow Patterns

Dans cette section, nous vérifions pour chaque pattern de données si leur implémentation est possible. Ces patterns ont été élaborés par [Russell *et al.*, 2004a].

6.3.1 Data visibility

Ces patterns vérifient pour chaque contexte la construction et l'utilisation des données.

Pattern 1 : Task Data

Une tâche peut définir des variables locales et ses données sont accessibles seulement par cette tâche.

Implémentation

Dans le fichier XML de modélisation, une variable locale peut être déclarée de deux manières différentes.

Soit on déclare la variable locale à l'aide du tag *localVariable* que nous avons expliqué à la figure 6.2. Ce type de déclaration n'est possible que dans une tâche composée. On utilisera ce type de variables pour stocker le résultat entre deux tâches ou pour stocker le résultat d'un XQuery. Voici la déclaration de la variable locale x dans le fichier XML :

```
<localVariable>
    <name>x</name>
    <type>string</type>
    <namespace>http://www.w3.org/2001/XMLSchema</namespace>
    <initialValue>n/a</initialValue>
</localVariable>
```

Soit on déclare la variable locale à l'aide du tag *inputParam* que nous avons expliqué à la figure 6.3. Cette variable est un paramètre d'entrée de la tâche. Voici la déclaration de la variable locale x dans le fichier XML :

```
<inputParam>
    <name>x</name>
    <type>string</type>
    <namespace>http://www.w3.org/2001/XMLSchema</namespace>
</inputParam>
```

Pattern 2 : Block Data

Une tâche composée peut définir des variables locales et ses données sont accessibles à toutes ses sous-tâches.

Implémentation

Dans le fichier XML de modélisation, la tâche composée déclare une variable locale. Chaque sous-tâche qui souhaite utiliser cette variable déclare cette variable en paramètre d'entrée. La déclaration au niveau de la tâche composée et de la sous-tâche s'effectue comme expliqué dans le pattern 1 (*Task Data*).

Il faut également indiquer le mapping à effectuer au démarrage de la tâche. Il faut prendre la variable du parent et la copier dans la variable d'entrée de la sous-tâche. Voici la déclaration du mapping de la variable locale x du parent vers le paramètre d'entrée x de la sous-tâche.

```
<mapping>
    <expression query="&lt;x&gt;{/parent/x/text()}&lt;/x&gt;" />
    <mapsTo>x</mapsTo>
</mapping>
```

Pattern 3 : Scope Data

On peut définir des données qui sont accessibles par un sous-ensemble du workflow.

Implémentation

La donnée est en fait une variable locale de la tâche composée *root*. Nous utilisons le mécanisme expliqué au pattern 2 (*Block Data*).

Pattern 4 : Multiple Instance Data

Une instance de tâche peut définir des variables locales et ses données sont accessibles seulement à cette instance et pas à d'autres instances de la même tâche.

Implémentation

Tous les paramètres d'une instance d'une tâche lors de l'exécution sont accessibles seulement à celle-ci. Une tâche comme on l'a vu dans le perspective précédente est un *thread* Java. Toutes les tâches sont donc indépendantes l'une de l'autre.

Pattern 5 : Case Data

Les données sont accessibles par tous les composants du workflow pendant l'exécution d'un cas.

Implémentation
La variable est en fait une variable locale de la tâche composée *root*.

Pattern 6 : Workflow Data

Les données sont accessibles par tous les composants de chaque instance du workflow. Ici, les données sont donc globales au workflow.

Implémentation
Nous rajoutons un tag à la déclaration d'une variable afin de savoir si cette variable doit être accessible à toutes les instances de workflow. Voici la déclaration de la variable locale x dans le fichier XML :

```
<localVariable>
    <name>x</name>
    <type>string</type>
    <namespace>http://www.w3.org/2001/XMLSchema</namespace>
    <initialValue>n/a</initialValue>
    <static>true</true>
</localVariable>
```

Nous déclarons un nouveau tag *static*. Si ce tag est présent, le moteur de workflow va stocker la variable dans un espace de tuples au lieu du fichier XML. Nous aurons un espace de tuples au cœur du moteur de workflow qui reprend les variables qui doivent être accessibles à tous les composants de chaque instance du workflow.

Pour être complet, il faut aussi bien entendu modifier le schéma XML qui valide le fichier YAWL. Nous avons donc rajouté un élément à la définition du tag *localVariable*. Voici la définition de cet élément :

```
<xs:element name="static" type="xs:boolean" minOccurs="0" />
```

Pattern 7 : Environment Data

Les données se trouvant dans un autre système (extérieur au moteur de workflow) peuvent être accédées par les composants du workflow pendant leur exécution.

Implémentation
Nous utilisons un web service pour communiquer avec l'extérieur. Le mécanisme de web service sera expliqué plus en détails dans la perspective opérationnelle.

Nous devons indiquer, au niveau de la donnée, où se trouve le fichier WSDL du web service afin de pouvoir l'appeler. Par exemple, voici la déclaration d'une variable locale x :

```
<localVariable>
    <name>x</name>
    <type>string</type>
    <namespace>http://www.w3.org/2001/XMLSchema</namespace>
    <initialValue>n/a</initialValue>
    <wsdl>http://test.wsdl</wsdl>
</localVariable>
```

Nous déclarons donc un noueau tag *wsdl*. Si ce tag est présent, le moteur de workflow va utiliser un web service pour aller chercher le contenu de la variable. Ce contenu est stocké dans un système externe au moteur de workflow.

Comme pour le pattern 6 (*Workflow Data*), nous rajoutons un élément au tag *localVariable* dans le schéma XML. Voici la définition de cet élément :

```
<xs:element name="wsdl" type="xs:string" minOccurs="0" />
```

6.3.2 Data Interaction

Cette catégorie reprend les patterns qui sont liés aux interactions des données. C'est-à-dire, la façon dont les données transitent dans le workflow et l'influence que ce transit peut avoir sur celui-ci.

Pattern 8 : Data Interaction - Task to Task

La capacité de communiquer des données entre une tâche et une autre dans le même workflow.

Implémentation
Notre fichier de modélisation contient tous les *mappings* possibles pour les entrées et sorties de chaque tâche. La communication d'une donnée d'une tâche à une autre tâche s'effectue en passant par le parent de ces deux tâches.

Au début d'une tâche, on effectue les mappings qui se trouvent dans la tag *startingMappings* que nous avons expliqué avec la figure 6.4. Ceci a pour effet de copier les valeurs des variables du parent vers la sous-tâche.

À la fin d'une tâche, on effectue les mappings qui se trouve dans le tag *completeMappings* que nous avons expliqué avec la figure 6.4. Ceci a pour effet de copier les valeurs de la sous-tâche dans la tâche parent.

Pattern 9 : Data Interaction - Block Task to Sub-Workflow Decomposition

La capacité de communiquer des données d'une tâche à ses sous-tâches.

Implémentation
L'implémentation est identique au pattern précédent. En effet, dans notre implémentation, un workflow est représenté par une tâche composée. C'est pourquoi, ici, les patterns 8 et 9 sont identiques. La description de la sous-tâche décrit les mappings à effectuer avec les variables locales du parent et les paramètres d'entrées de la sous-tâche.

Pattern 10 : Data Interaction - Sub-Workflow Decomposition to Block Task

La capacité de communiquer des données de sous-tâches d'une tâche composée à sa tâche parente.

Implémentation
L'implémentation est identique au pattern précédent mais ici pour les paramètres de sortie.

Pattern 11 : Data Interaction - to Multiple Instance Task

La capacité de passer des données d'une tâche à une sous-tâche qui est capable de supporter l'exécution d'instances multiples.

Implémentation
Nous utilisons une expression XQuery qui se trouve dans le fichier de modélisation. Voici un exemple de XQuery que nous avons repris de notre fichier de modélisation pour notre exemple du premier chapitre :

```
<miDataInput>
    <expression query="/make_trip/legs" />
    <splittingExpression query="for $d in /legs/* return $d" />
    <formalInputParam>leg</formalInputParam>
</miDataInput>
```

Cette expression permet de faire une boucle sur les différents itinéraires que le client a choisi (*leg*). Ces itinéraires sont stockés dans la variable *leg* de la tâche composée `make_trip`. Pour chaque itinéraire, nous allons créer un nouveau fichier XML par instance de la tâche. Le contenu de l'itinéraire est stocké dans le variable *leg* de chaque instance de la sous-tâche.

Pattern 12 : Data Interaction - from Multiple Instance Task

Ce pattern est identique au précédent sauf qu'ici les données proviennent des instances multiples d'une tâche.

Implémentation
Nous utilisons une expression XQuery qui se trouve dans le fichier de modélisation. Voici également un exemple que nous avons repris de notre fichier de modélisation pour notre exemple du premier chapitre :

```
<miDataOutput>
  <formalOutputExpression query=
    "<itinerarySegment>
      { /do_itinerary_segment/leg/departure_location }
      { /do_itinerary_segment/leg/destination }
      { /do_itinerary_segment/startDate }
      { /do_itinerary_segment/endDate }
      { if(/do_itinerary_segment/flightDetails/text()) then
        /do_itinerary_segment/flightDetails else ()}
      { if(/do_itinerary_segment/hotelDetails/text()) then
        /do_itinerary_segment/hotelDetails else() }
            { if(/do_itinerary_segment/carDetails/text()) then
        /do_itinerary_segment/carDetails else() }
            { /do_itinerary_segment/subTotal }
```

```
      </itinerarySegment>"
    />
    <outputJoiningExpression query="<itinerary>
      {for \$d in /do_itinerary_segment/itinerarySegment return
      \$d}</itinerary>" />
    <resultAppliedToLocalVariable>itinerary</resultAppliedToLocalVariable>
  </miDataOutput>
```

Cette expression permet de boucler sur chaque instance de la tâche (`/do_itinerary_segment/itinerarySegment`) et de rassembler l'information dans une variable de la tâche parente. Dans notre exemple, nous stockons le résultat dans la variable *itinerary*. Dans le tag *formalOutputExpression*, on construit la variable *itinerary* pour chaque instance à partir des variables locales de l'instance de la sous-tâche.

Pattern 13 : Data Interaction - Case to Case

La capacité de passer des données d'un cas du workflow à un autre cas du workflow qui s'exécute de façon concomitante.

Implémentation
Nous utilisons le pattern 6 (*Workflow Data*) pour passer certaines données d'une instance de workflow à une autre instance de workflow.

6.3.3 Data Interaction - External data passing

Cette section regroupe les patterns qui interagissent avec des systèmes externes au moteur de workflow. Nous utilisons un web service pour communiquer avec l'extérieur et celui-ci sera expliqué plus en détails dans la perspective opérationnelle. Nous ne citons ici que le pattern et sa description.

Pattern 14 : Data Interaction - Task to Environment - Push-Oriented
La capacité d'une tâche à initier le passage des données à une ressource (ou service) extérieure.

Pattern 15 : Data Interaction - Environment to Task - Pull-Oriented
La capacité d'une tâche à demander des données d'une ressource (ou service) extérieure.

Pattern 16 : Data Interaction - Environment to Task - Push-Oriented
La capacité d'une tâche de recevoir et d'utiliser des données reçues d'une ressource (ou service) extérieure de façon non planifiée. C'est-à-dire, qu'elle peut recevoir ces données à tout moment et qu'elle ignore le moment de la réception.

Pattern 17 : Data Interaction - Task to Environment - Pull-Oriented
La capacité d'une tâche de recevoir et traiter des données provenant de ressource (ou service) extérieure.

Pattern 18 : Data Interaction - Case to Environment - Push-Oriented
La capacité d'un cas du workflow à initier le passage des données à une ressource (ou service) extérieure.

Pattern 19 : Data Interaction - Environment to Case - Pull-Oriented
La capacité d'un cas du workflow à demander des données d'une ressource (ou service) extérieure.

Pattern 20 : Data Interaction - Environment to Case - Push-Oriented
La capacité d'un cas du workflow de recevoir et d'utiliser des données reçues d'une ressource (ou service) extérieure de façon non planifiée. C'est-à-dire, qu'elle peut recevoir ces données à tout moment et qu'elle ignore le moment de la réception.

Pattern 21 : Data Interaction - Case to Environment - Pull-Oriented
La capacité d'un cas du workflow de recevoir et traiter des données provenant de ressource (ou service) extérieure.

Pattern 22 : Data Interaction - Workflow to Environment - Push-Oriented
La capacité d'un moteur de workflow de passer des données à des ressources (ou service) extérieures.

Pattern 23 : Data Interaction - Environment to Workflow - Pull-Oriented
La capacité d'un moteur de workflow à demander des données à des ressources (ou service) extérieures.

Pattern 24 : Data Interaction - Environment to Workflow - Push-Oriented
La capacité d'une ressource (ou service) extérieure à passer des données au workflow. Ces données seront stockées au niveau du workflow (portée globale).

Pattern 25 : Data Interaction - Workflow to Environment - Pull-Oriented
La capacité d'un moteur de workflow de traiter des requêtes provenant de ressources (ou services) extérieures pour des données globales.

6.3.4 Data Transfer Mechanisms

Ces patterns décrivent la façon dont les données sont transférées. Les différents mécanismes de transfert s'inspirent fortement de ceux utilisés dans les langages de programmation lorsqu'on effectue un appel de méthode. Ainsi, on retrouvera le passage de données par valeur et par référence.

Pattern 26 : Data Transfer by Value - Incoming

La capacité des éléments du workflow à recevoir des données entrantes par valeur. Ceci est le paradigme habituel du « passage par valeur » dans les langages de programmation.

Implémentation
Si nous avions choisi ce mode de transfert, nous aurions nos données dans le jeton de contrôle que nous avions vu dans la perspective précédente. Quand la tâche reçoit le contrôle, elle récupère en même temps les valeurs des variables.

Pattern 27 : Data Transfer by Value - Outgoing

La capacité des éléments du workflow à passer des données (sortantes) par valeur.

Implémentation
Similaire au pattern précédent, quand la tâche est finie, elle copie les valeurs des variables dans le jeton de contrôle.

Pattern 28 : Data Transfer - Copy In/Copy Out

La capacité d'un élément du workflow à copier un ensemble de données dans son espace mémoire (ou espace d'adresses) au début de son exécution (*copy in*). À la fin de son exécution, il copie les valeurs finales de ses données dans leur espace mémoire d'origine (*copy out*).

Implémentation
Pour passer les données d'une tâche à une autre tâche, nous passons par un espace de stockage commun qui se trouve au niveau du parent. Au début d'une tâche, on va chercher la valeur de la variable dans son parent. À la fin d'une tâche, on copie la valeur de la variable dans son parent.

Pattern 29 : Data Transfer by Reference - Unlocked

La capacité des éléments du workflow à passer les données par référence. Dans le sens où ces données sont stockées dans un espace mémoire partagé par les deux composants qui désirent communiquer. Notons aussi qu'on ne tient pas compte des accès concomitants (voir pattern suivant).

Implémentation
Ce pattern est identique au précédent mais au lieu de copier la valeur de la tâche à la sous-tâche, on copie la référence. Ceci permet d'avoir la valeur seulement stockée à un seul endroit

Pattern 30 : Data Transfer by Reference - With Lock

La capacité des éléments du workflow à passer les données par référence. Dans le sens où ces données sont stockées dans un espace mémoire partagé par les deux composants qui désirent communiquer. Ici, on tient compte des accès concomitants en utilisant des verrous sur les données.

Implémentation
Ce pattern est identique au précédent mais nous devons nous assurer que deux personnes ne modifient pas les données en même temps. Nous allons utiliser les mécanismes de synchronisation de Java.

Pattern 31 : Data Transformation - Input

La capacité d'appliquer une fonction de transformation aux données avant de les passer à la tâche.

Implémentation
Nous pouvons utiliser des opérateurs tels que la division, la multiplication, etc dans une expression XPath et XQuery. Voici un exemple qui permet de transformer un sous-total d'un format texte en format numérique afin de pouvoir faire une addition plus tard.

```
<expression query="<subTotal>{number(/make_trip/subTotal/text())}</subTotal>" />
```

Pattern 32 : Data Transformation - Output

La capacité d'appliquer une fonction de transformation aux données avant de les envoyer à la tâche suivante.

Implémentation
Ce pattern est identique au pattern précédent.

6.3.5 Data-based Routing

Ces patterns décrivent la façon dont les données peuvent influencer les autres perspectives et la mise en œuvre globale du workflow.

Pattern 33 : Task Precondition - Data Existence

Pour ce pattern, on peut spécifier qu'une tâche s'exécute si et seulement si une donnée existe lors de l'exécution du workflow. C'est-à-dire qu'on définit une pré-condition sur l'exécution de la tâche.

Implémentation
On peut implémenter ce pattern de plusieurs façons différentes. En effet, lorsqu'une donnée est manquante, on peut :
- attendre jusqu'à ce que la donnée soit disponible ;
- spécifier des valeurs par défaut ;
- demander la valeur des données interactivement ;
- ne pas exécuter la tâche ;
- annuler l'exécution de ce cas du workflow.

Pour notre implémentation, nous avons choisi de lancer une exception dès qu'une donnée est manquante. On considère donc l'absence de donnée comme une erreur dans le workflow. Néanmoins, pour éviter de faire du traitement d'exceptions dans chaque tâche de notre workflow, on va convenir d'un traitement standard de cette exception. Ce traitement standard pourrait être par exemple de demander la valeur de ces données interactivement à l'utilisateur. On aurait pu aussi utiliser des valeurs par défaut. Ceci n'est qu'un choix d'implémentation.

Pattern 34 : Task Precondition - Data Value

Pour ce pattern, on peut spécifier qu'une tâche s'exécute si et seulement si un ensemble de données possède certaines valeurs lors de l'exécution du workflow.

Implémentation
Il y a une possibilité de mettre une condition d'entrée à une tâche. Nous utilisons une expression XPath et cette condition est encodée dans le tag *flowInto*. Dans notre exemple, nous irons à la tâche suivante que si la variable ok dans le parent est vraie.

```
<flowsInto>
    <nextElementRef id="suivant" />
    <predicate>/parent/ok='true'</predicate>
</flowsInto>
```

Pattern 35 : Task Postcondition - Data Existence

Ce pattern est identique au pattern 33 sauf qu'on définit ici l'existence des données en tant que post-condition.

Implémentation
Même implémentation qu'au pattern 33.

Pattern 36 : Task Postcondition - Data Value

Ce pattern est identique au pattern 34 sauf qu'on définit ici la valeur des données en tant que post-condition.

Implémentation
Même implémentation qu'au pattern 34.

Pattern 37 : Event-based Task Trigger

La capacité d'un événement extérieur au moteur de workflow d'initier une tâche.

Implémentation
Toute tâche de notre moteur de workflow peut être lancée par un web service. Nous verrons leur implémentation dans la perspective opérationnelle.

Pattern 38 : Data-based Task Trigger

La capacité de déclencher une tâche lorsque une expression (basée sur les données du workflow) s'avère vraie dans le workflow.

Implémentation
Même implémentation qu'au pattern 34

Pattern 39 : Data-based Routing

La capacité de changer le flux du contrôle du workflow dans une instance du workflow selon la valeur des données.

Implémentation
Même implémentation qu'au pattern 34

6.4 Conclusion

Dans ce chapitre, nous avons d'abord commencé en complétant l'explication de la structure de notre fichier XML pour la manipulation des données du workflow. Ensuite, nous avons dû choisir entre deux implémentations possibles. Enfin, nous avons vérifié si chaque pattern de données pouvait être implémenté à partir de notre choix d'implémentation [Russell *et al.*, 2004a]. Nous ne sommes pas rentrés dans les détails car le but de ce chapitre est de comprendre le principe, d'effectuer un choix d'implémentation et de vérifier si celle-ci est possible.

Dans le chapitre suivant, nous analysons et implémentons la perspective de ressources.

Chapitre 7

Perspective de ressources

Une ressource est toute entité capable d'accomplir une tâche. Une ressource peut donc être une personne mais aussi un programme extérieur.

Dans ce chapitre, nous analysons la gestion des ressources au sein du moteur de workflow. Nous commençons notre analyse par le parcours de tous les patterns associés à cette perspective. Ensuite, on présente les scénarios d'utilisation et le diagramme de classes qui résultent de l'analyse des patterns. Finalement, on présente les différentes modifications nécessaires au sein de notre architecture pour pouvoir implémenter cette perspective.

7.1 Parsing du fichier XML

Le fichier YAWL reçu par le moteur de workflow en entrée ne prend pas en compte la gestion des ressources. Dès lors, nous ne présentons pas ici le parsing du fichier XML. Néanmoins, nous aurons besoin de ces informations in fine. Nous avons donc décidé d'analyser dans un premier temps les besoins nécessaires pour cette perspective. Pour cette analyse, on utilise les patterns identifiés dans la littérature comme dans les perspectives précédentes. Puis, dans un second temps, nous présentons les changements que nous estimons nécessaires au niveau du fichier XML.

7.2 Resource Pattern

Dans cette section, nous analysons les patterns de ressources afin de connaître les différentes règles de gestion des ressources au sein d'un workflow. Ces patterns ont été élaborés par [Russell *et al.*, 2004b].

La figure 7.1 représente le cycle de vie d'une tâche atomique sous la forme d'un diagramme d'états. Le cycle de vie commence à la création et se termine une fois que la tâche est accomplie ou annulée.

Nous remarquons sur les transitions que le nom est préfixé soit d'un S soit d'un R indiquant que la transition est initiée soit par le moteur de workflow soit par la ressource.

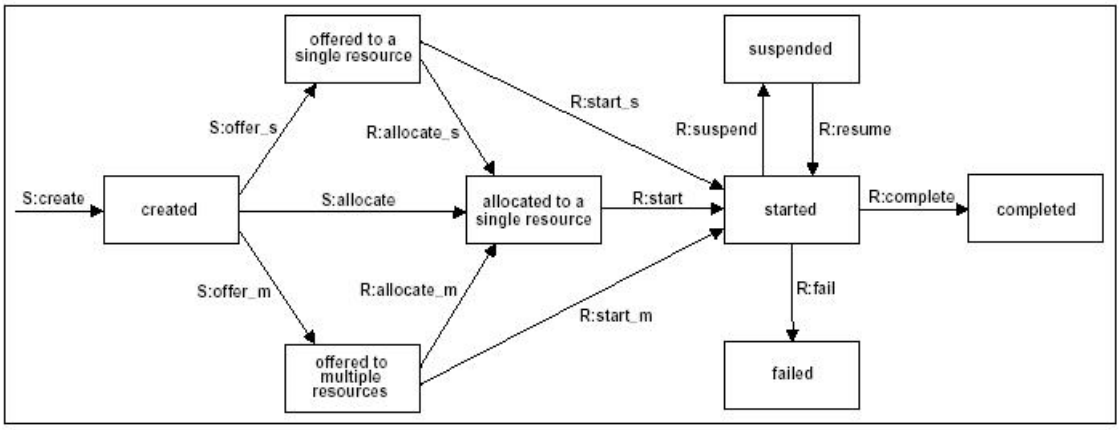

Figure 7.1 – Cycle de vie d'une tâche [Russell *et al.*, 2004b]

7.2.1 Creation Patterns

Cette section présente des patterns qui correspondent à la transition « *S :create* » de la figure 7.1. Elle est initiée par le moteur de workflow. Une fois que la tâche est créée, la tâche se retrouve dans l'état *created*.

Nous retrouvons dans ces patterns l'attribution d'une ressource pour une tâche.

Pattern 1 : Direct Allocation
Lors de la conception du workflow, on assigne la tâche à une personne bien précise.

Pattern 2 : Role-Based Allocation
Lors de la conception du workflow, on assigne la tâche à toutes les personnes d'un certain rôle. Lorsque la première personne accepte la tâche, celle-ci est supprimée chez les autres personnes du groupe.

Pattern 3 : Deferred Allocation
Lors de l'exécution, on demande le nom de la ressource qu'on souhaite pour exécuter la tâche et puis elle est assignée à cette ressource.

Pattern 4 : Authorisation
On assigne la tâche à un groupe de personnes qui est autorisé à exécuter la tâche.

Pattern 5 : Separation of Duties
On peut indiquer que deux ou plusieurs tâches doivent être assignées à différentes ressources dans un même workflow. Ceci est utile quand une personne doit effectuer un travail et que la tâche suivante est la vérification de ce travail. Il est évident que les deux personnes assignées à ces tâches ne peuvent pas être la même.

Pattern 6 : Case Handling
On peut indiquer que deux ou plusieurs tâches doivent être assignées à la même personne dans un même workflow. Ceci est utile quand une personne doit préparer un travail et que ça doit être la même personne qui présente ce travail.

Pattern 7 : Retain Familiar
Quand plusieurs ressources sont disponibles pour entreprendre une tâche, celle-ci est assignée à la même ressource qui a effectué la tâche précédemment.

Pattern 8 : Capability-based Allocation
La tâche est offerte ou assignée à une ressource sur base de sa capacité. Par exemple, sur base de son diplôme.

Pattern 9 : History-based Allocation
La tâche est offerte ou assignée à la ressource qui a déjà effectué le plus souvent cette tâche.

Pattern 10 : Organisational Allocation
La tâche est offerte ou assignée à la ressource sur base de sa position dans l'organisation et de ses rapports avec les autres ressources.

Pattern 11 : Automatic Execution
La tâche s'exécute sans devoir utiliser les services d'une ressource.

7.2.2 Push Patterns

Cette section présente des patterns qui correspondent à la transition « *S :offer_s* », « *S :offer_m* » et « *S :allocate* » de la figure 7.1. Elles sont initiées par le moteur de workflow.

Nous retrouvons dans ces patterns le fait qu'une tâche est offerte ou assignée aux ressources par le système de workflow. Il y a trois types d'attributions.

S :offer_s La tâche est attribuée à une ressource et la tâche arrive dans l'état *offered to a single resource*.

S :offer_m La tâche est offerte à différentes ressources mais seulement une ressource effectuera la tâche. La tâche arrive dans l'état *offered to multiple resources*.

S :allocate La tâche est directement attribuée à une ressource après la création de la tâche. La tâche arrive dans l'état *allocated to a single resource*.

Ces patterns permettent donc d'offrir ou d'assigner aux ressources des tâches.

Pattern 12 : Distribution by Offer - Single Resource
Une tâche est offerte à une seule ressource.

Pattern 13 : Distribution by Offer - Multiple Resource
Une tâche est offerte à un groupe de ressources. La première ressource qui accepte la tâche la reçoit et elle est enlevée aux autres personnes du groupe.

Pattern 14 : Distribution by Allocation - Single Resource
Une tâche est directement assignée à une ressource spécifique pendant l'exécution.

Pattern 15 : Random Allocation
Une tâche est offerte ou assignée aux ressources appropriées de manière aléatoire.

Pattern 16 : Round Robin Allocation
Une tâche est assignée aux ressources disponibles sur une base cyclique.

Pattern 17 : Shortest Queue
Une tâche est assignée à la ressource qui a le moins de travail en cours et qui est capable d'effectuer cette tâche.

Pattern 18 : Early Distribution
Une tâche est annoncée et assignée potentiellement aux ressources avant que la tâche soit permise réellement.

Pattern 19 : Distribution on Enablement
Une tâche est annoncée et assignée aux ressources au moment où elle est permise pour l'exécution.

Pattern 20 : Late Distribution
Une tâche est annoncée et assignée aux ressources après que la tâche ait été permise.

7.2.3 Pull Patterns

Cette section présente des patterns qui correspondent à la transition « *R :allocate_s* », « *R :allocate_m* », « *R :start_s* », « *R :start_m* » et « *R :start* » de la figure 7.1. Elles sont initiées par les ressources.

Nous retrouvons dans ces patterns le fait qu'une tâche est offerte ou assignée aux ressources par la ressource elle-même. Il y a cinq types d'attributions.

R :allocate_s La tâche a été offerte à une ressource et la ressource a accepté d'effectuer le travail. La tâche arrive dans l'état *allocated to a single ressource*.

R :allocate_m La tâche a été offerte à un groupe de ressources et une ressource a accepté d'effectuer le travail. La tâche arrive dans l'état *allocated to a single ressource*.

R :start_s La tâche a été offerte à une ressource et la ressource a commencé à effectuer le travail. La tâche arrive dans l'état *started*.

R :start_m La tâche a été offerte à un groupe de ressources et une ressource a commencé à effectuer le travail. La tâche arrive dans l'état *started*.

R :start La tâche a été assignée à une ressource et la ressource a commencé à effectuer le travail. La tâche arrive dans l'état *started*.

Nous distinguons dans les patterns de cette section deux groupes. Les trois premiers patterns correspondent aux transitions de la figure 7.1. Les trois derniers représentent l'ordre dans lequel les tâches d'une ressource sont présentées et la capacité à chacun d'influencer l'ordre. Ces patterns n'ont pas d'analogue direct aux transitions de la figure 7.1 mais s'appliquent à toutes les transitions que nous venons d'analyser.

Pattern 21 : Resource-Initiated Allocation
Une tâche est acceptée par une ressource mais ne doit pas débuter immédiatement. La tâche est placée dans la liste des tâches allouées.

Pattern 22 : Resource-Initiated Execution - Allocated Work Item
Une tâche est commencée par une ressource si celle-ci lui est assignée.

Pattern 23 : Resource-Initiated Execution - Offered Work Item
Une tâche est choisie par une ressource et elle débute directement.

Pattern 24 : System-Determined Work Queue Content
Le moteur de workflow présente le contenu et l'ordre des différentes tâches par ressource.

Pattern 25 : Resource-Determined Work Queue Content
Les ressources indiquent le format et le contenu des tâches dans la liste de travail.

Pattern 26 : Selection Autonomy
Une ressource choisit sur base de ses caractéristiques et de ses préférences une tâche à exécuter dans sa liste de tâches.

7.2.4 Detour Patterns

Cette section présente des patterns pour la réallocation de tâches au cas où la ressource précédente ne peut pas effectuer le travail. Par conséquent, l'ordre normal des transitions pour une tâche va changer. La figure 7.2 illustre les différents scénarios possibles en indiquant les transitions en gras.

Il y a un certain nombre d'impacts possibles sur une tâche, selon son état actuel de progression et si l'action a été lancée par la ressource ou par le système de workflow. Chacune de ces transitions est associée à un pattern et un pattern peut décrire une ou plusieurs transitions.

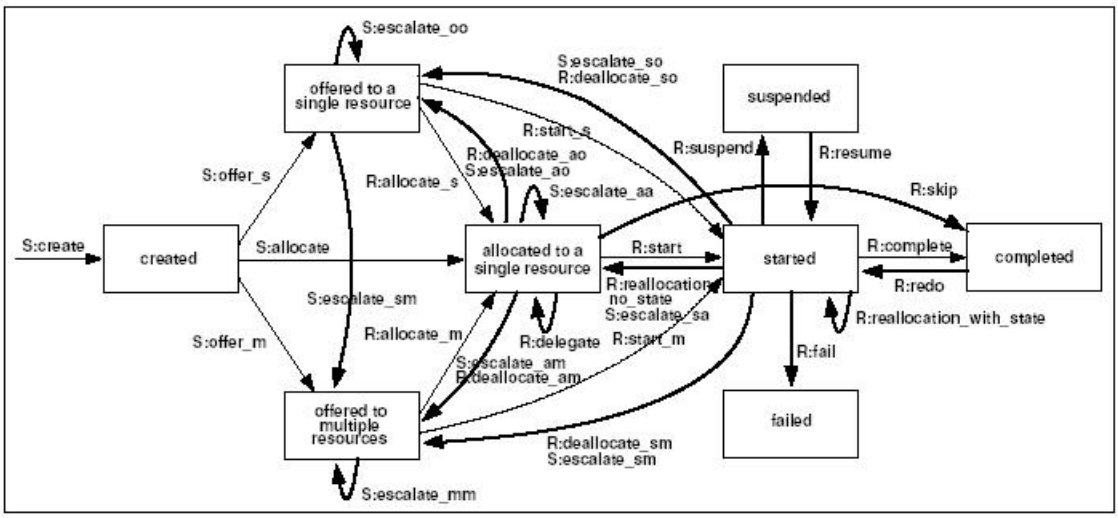

Figure 7.2 – Detour Patterns [Russell *et al.*, 2004b]

Pattern 27 : Delegation
Une tâche assignée à une ressource peut être assignée à une autre ressource.

Pattern 28 : Escalation
Le moteur de workflow offre ou alloue une tâche à une ressource ou à un groupe de ressources après que la ressource précédente ait dépassé le temps valide pour exécuter la tâche.

Pattern 29 : Deallocation
Une ressource ou un groupe de ressources abandonne une tâche qui lui est assignée et la rend disponible pour l'attribuer à une autre ressource ou groupe de ressources.

Pattern 30 : Stateful Reallocation
Une ressource réassigne une tâche qu'elle a déjà commencé à une autre ressource sans perdre les données déjà associées à la tâche.

Pattern 31 : Stateless Reallocation
Une ressource réassigne la tâche qu'elle a déjà commencé à une autre ressource sans conserver les données déjà associées à la tâche.

Pattern 32 : Suspension-Resumption
Une ressource peut suspendre et reprendre une tâche lors de son exécution.

Pattern 33 : Skip
Une ressource peut sauter une tâche qui lui est assignée et indiquer la tâche comme effectuée.

Pattern 34 : Redo
Une ressource peut réexécuter une tâche qui a été précédemment accomplie.

Pattern 35 : Pre-Do
Une ressource peut exécuter une tâche avant qu'elle ne lui soit offerte ou assignée.

7.2.5 Auto-start Patterns

Cette section présente des patterns qui correspondent aux transitions qui sont en gras dans la figure 7.3.

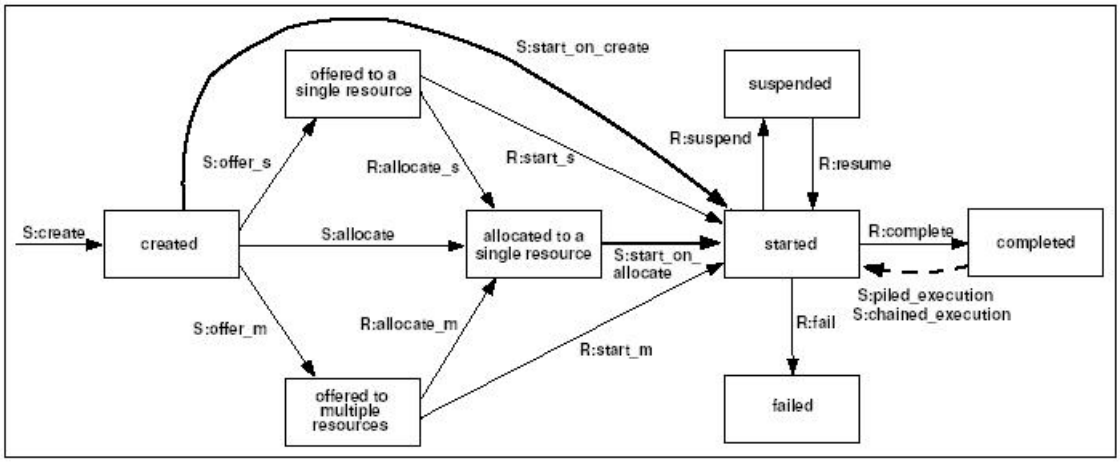

Figure 7.3 – Auto-start Patterns [Russell *et al.*, 2004b]

Ces transitions sont déclenchées par des événements spécifiques, par exemple l'assignation d'une tâche à une ressource, dans le cycle de vie de la tâche. Chacune de ces transitions est associée à un pattern.

Pattern 36 : Commencement on Creation
Une ressource peut débuter l'exécution d'une tâche dès qu'elle est créée.

Pattern 37 : Commencement on Allocation
La tâche est débutée dès qu'elle est assignée à une ressource.

Pattern 38 : Piled Execution
Le moteur de workflow lance les prochaines instances d'une tâche une fois que la précédente est accomplie.

Pattern 39 : Chained Execution
Le moteur de workflow commence automatiquement la prochaine tâche une fois que la précédente est accomplie.

7.2.6 Visibility Patterns

Cette section présente les patterns de visibilité qui classifient les diverses portées dans lesquelles la tâche peut être vue.

Pattern 40 : Configurable Unallocated Work Item Visibility
On peut configurer la visibilité des tâches non affectées à des participants du moteur de workflow.

Pattern 41 : Configurable Allocated Work Item Visibility
On peut configurer la visibilité des tâches affectées à des participants du moteur de workflow.

7.2.7 Multiple Resource Patterns

Cette section présente les patterns qui permettent à une ressource de pouvoir exécuter plusieurs tâches de sa liste de travail.

Pattern 42 : Simultaneous Execution
Une ressource peut exécuter plusieurs tâches simultanément.

Pattern 43 : Additional Resources
Une ressource peut demander de l'aide à l'exécution d'une tâche qu'elle entreprend actuellement.

7.3 Scénario d'utilisation

Nous présentons les différents scénarios d'utilisation possibles en ce qui concerne la gestion des ressources dans notre moteur de workflow.

7.3.1 Moteur de workflow

Le scénario d'utilisation de la figure 7.4 montre les différentes possibilités que le moteur de workflow a à sa disposition pour gérer la gestion des ressources. Le moteur de workflow peut donc offrir la tâche à une ressource (*Offer a task to a resource*), offrir la tâche à un groupe de ressources (*Offer a task to a group*) et réassigner une tâche (*escalate a task*).

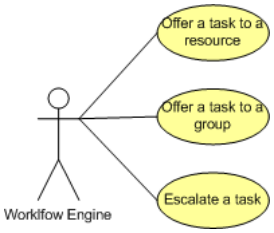

Figure 7.4 – Scénario d'utilisation du moteur de workflow

7.3.2 Ressource

Le scénario d'utilisation de la figure 7.5 montre les différentes possibilités qu'une ressource du moteur de workflow a à sa disposition. La ressource peut donc accepter la tâche(*Allocate a task*), commencer une tâche(*start a task*), suspendre une tâche (*suspend a task*), relancer une tâche (*resume a task*), annuler la tâche (*fail a task*), terminer la tâche (*complete a task*), annuler une tâche mais la considérer comme terminée (*skip a task*), refaire une tâche (*redo a task*), réassigner une tâche (*reallocate a task*) et abandonner une tâche (*deallocate a task*).

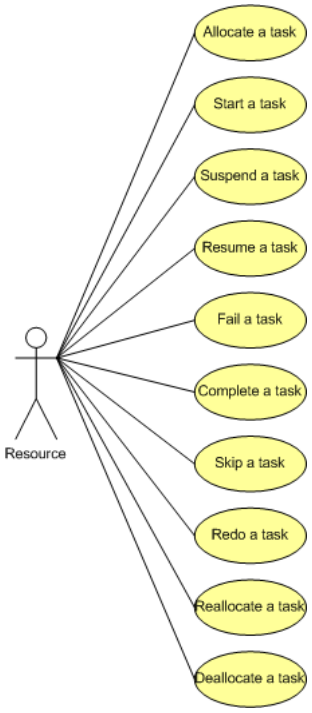

Figure 7.5 – Scénario d'utilisation pour une ressource

7.4 Persistance des ressources

Après avoir analysé les différents patterns de cette perspective, nous avons décidé de séparer les informations liées aux ressources en trois parties distinctes :

- Annuaire de l'entreprise ;
- Modifications du fichier XML ;
- Persistance de l'état du moteur de workflow.

7.4.1 Annuaire de l'entreprise

L'annuaire de l'entreprise contient toutes les informations de chaque ressource. Ces informations sont nécessaires au moteur de workflow mais sont aussi utilisées par les autres processus de l'entreprise. Le contenu typique d'un annuaire contiendra les informations suivantes :

idResource L'identifiant de la ressource dans la société.

nom Le nom de la ressource.

prenom Le prénom de la ressource.

role Le rôle de la ressource dans la société.

groupe Le groupe dans lequel la ressource travaille. Par exemple : informaticien.

service Le service dans lequel la ressource travaille. Par exemple : service informatique.

diplome Le ou les diplômes de la ressource.

description La description des capacités de la ressource.

Notons cependant que cet annuaire est spécifique pour chaque entreprise. C'est pourquoi, nous n'allons pas encoder ces informations dans notre moteur de workflow. Par exemple, dans un workflow d'approbation de factures, chaque ressource peut approuver des factures. Néanmoins, selon le rang de la ressource dans l'entreprise, le montant des factures peut être plus ou moins élevé. Supposons que David a le rang C et Michel le rang B, David peut approuver des factures dont le montant est inférieur à 10 000 EUR. Quant à Michel, il peut approuver des factures dont le montant est inférieur à 100 000 EUR. Seul, le PDG de l'entreprise peut approuver des factures plus élevées car il est de rang A.

Donc, si notre annuaire ne contient pas de rang, on ne sait pas implémenter ces règles dans notre moteur de workflow. Toutefois, comme ces règles sont spécifiques à une entreprise, on ne doit surtout pas encoder le rang dans notre moteur de workflow.

7.4.2 Modifications du fichier XML

Comme mentionné au début du chapitre, le fichier YAWL en tant que tel ne contient aucune information concernant les ressources. On devra donc forcément le modifier si on désire traiter les ressources dans notre moteur de workflow. Nous proposons néanmoins d'inclure un minimum d'informations sur les ressources dans ce fichier. Comme nous l'avons déjà expliqué, une grande partie de celles-ci se trouvent déjà dans l'annuaire de l'entreprise et n'ont donc pas besoin d'être répétées.

Ce qui ne se trouve pas dans l'annuaire sont les informations spécifiques au workflow, tels que par exemple quelle ressource peut effectuer ou accéder à une tâche. On doit aussi inclure les actions possibles qui découlent des patterns que nous avons étudiés. Pour cela, nous avons décidé d'assigner une action possible par tâche. Il y a donc une relation 1-à-1 entre chaque tâche du workflow et l'ensemble des actions possibles.

De façon pragmatique, voici donc les changements proposés au niveau du fichier XML :
- chaque tâche se voit assigner une action (parmi les actions possibles) et des paramètres afin de la configurer si nécessaire.
- chaque tâche peut être associée à un ensemble de permissions (liste des ressources pouvant accéder à la tâche).
- chaque tâche peut définir un *timeout* afin de réassigner la tâche à une autre ressource si le temps est écoulé.

Voici un exemple d'un fichier XML pour déclarer une tâche dans le workflow.

```
<decomposition id="register" xsi:type="WebServiceGatewayFactsType">
    <inputParam>
        <name>customer</name>
        <type>string</type>
        <namespace>http://www.w3.org/2001/XMLSchema</namespace>
    </inputParam>

    [...]

</decomposition>
```

Voici le même exemple mais avec les changements nécessaires pour la gestion des ressources.

```
<decomposition id="register" xsi:type="WebServiceGatewayFactsType">
    <inputParam>
        <name>customer</name>
        <type>string</type>
        <namespace>http://www.w3.org/2001/XMLSchema</namespace>
    </inputParam>

    [...]

    <resourceManager>
        <action>
            <id>R_DA</id>
            <parameter>uid_123</parameter>
        </action>
        <timeout unit="d">2</timeout>
        <permission>uid_1, gid_12, uid_3</permission>
    </resourceManager>

</decomposition>
```

On remarque donc dans cet exemple que nous avons simplement ajouté des tags XML dans le tag décrivant la tâche. Pour être complet, il faut aussi bien entendu modifier le schéma XML qui valide le fichier YAWL. Nous avons donc rajouté les définitions suivantes dans lschéma XML de notre fichier de modélisation utilisé par YAWL :

```
<xs:complexType name="ResourceManagerType">
    <xs:sequence>
        <xs:element name="action" type="yawl:ActionType" minOccurs="0" />
        <xs:element name="permission" type="xs:string" minOccurs="0" />
        <xs:element name="timeout" type="yawl:TimeoutType" minOccurs="0" />
    </xs:sequence>
</xs:complexType>
<xs:complexType name="ActionType">
    <xs:sequence>
        <xs:element name="id" type="xs:string" />
        <xs:sequence>
            <xs:element name="parameter" type="xs:string" minOccurs="0" />
    </xs:sequence>
</xs:sequence>
</xs:complexType>
<xs:complexType name="TimeoutType">
    <xs:complexContent>
        <xs:restriction base="xs:int">
            <xs:attribute name="unit" type="xs:string" use="required" />
        </xs:restriction>
</xs:complexContent>
</xs:complexType>
```

Nous avons aussi ajouté une nouvelle entrée dans le tag *decomposition* (*yawl :DecompositionType*) en utilisant ce nouveau type :

```
<xs:element name="resourceManager" type="yawl:ResourceManagerType" minOccurs="0"
            maxOccurs="unbounded" />
```

7.4.3 Persistance de l'état du moteur de workflow

Remarquons que même en utilisant un annuaire externe et en modifiant le fichier XML d'entrée, cela ne suffit pas pour pouvoir implémenter tous les patterns présentés. En effet, certains patterns influencent la manière dont le moteur de workflow va s'exécuter.

Si on prend l'exemple du pattern *History-based Allocation*, celui-ci nécessite que l'on garde un historique des différentes ressources qui ont exécuté une tâche. Le moteur de workflow doit utiliser cet historique afin de choisir la ressource la plus appropriée lors de l'affectation de la tâche.

Pour implémenter ce pattern, on doit donc maintenir un historique. Néanmoins, cet historique ne peut être alimenté que par le moteur de workflow. Car lui seul sait quelle ressource a effectué la tâche. Avant et pendant l'exécution du workflow, la tâche peut toujours être réassignée à une autre ressource.

Pour accomplir la persistance de l'état du moteur de workflow, nous avons donc du modifier légèrement son architecture. Pour ce faire, nous avons ajouté deux classes Log et WorkList. La première permet de conserver les attributions des tâches pendant l'exécution du workflow. La seconde permet de présenter la liste des différentes tâches assignées à chaque ressource. Nous expliquons plus en détail ces deux classes dans la section suivante.

7.5 Modélisation du domaine d'application

La section précédente nous a permis de comprendre les différentes modifications nécessaires pour l'implémentation de la gestion des ressources dans notre moteur de workflow. Cette section présente une vue globale de ces modifications au niveau du moteur de workflow.

Le diagramme de classes de la figure 7.6 montre la modélisation de la perspective de gestion des ressources. Nous ne présentons ici que les classes qui concernent cette perspective afin de ne pas alourdir le diagramme de classes. La seule classe qui a été récupérée de la perspective de contrôle de flux est la classe taskAbstract.

On peut séparer à nouveau les changements du diagramme de classes en trois parties. La première partie est celle qui effectue le lien avec l'annuaire de l'entreprise. Pour réaliser ce lien, on utilise trois classes : Resource, User et Group. Toutefois, comme le contenu de l'annuaire est différent pour chaque entreprise, nous avons décidé de ne conserver qu'un identifiant. Cet identifiant sera unique dans l'annuaire de l'entreprise et dans notre moteur de workflow. Il identifiera aussi bien les utilisateurs que les groupes d'utilisateurs.

La deuxième partie concerne les changements effectués à notre fichier d'entrée. Dans celui-ci, on conserve l'action, le *time-out* et les permissions qui sont affectés à chaque tâche. Donc, pour modéliser ces changements, on conserve ces informations dans la classe TaskAbstract. De plus, on doit représenter toutes les actions possibles dans notre moteur de workflow. Chaque action sera modélisée par une classe. Évidemment, toutes les actions partagent des attributs communs que nous avons généralisés dans une classe abstraite ActionAbstract.

Enfin, la troisième partie concerne la persistance de l'état du moteur de workflow. Celle-ci est représentée par deux classes Log et WorkList. Celles-ci s'occupent de sauvegarder l'attribution des tâches aux ressources.

Nous définissons à présent chaque élément du diagramme de classes.

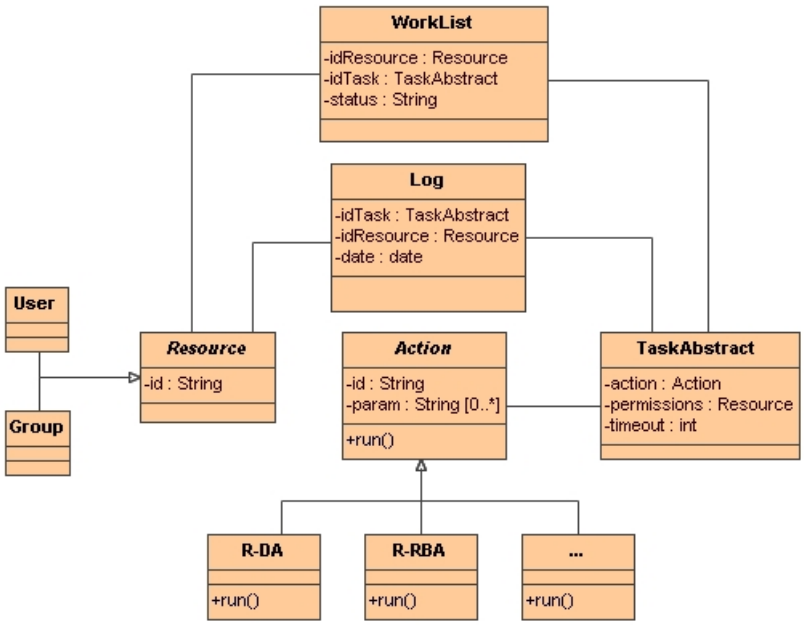

Figure 7.6 – Diagramme de classes

TaskAbstract

La classe abstraite TaskAbstract représente tous les attributs communs d'une tâche. Elle ne sera pas présente en tant que telle dans le workflow mais est la généralisation d'une tâche atomique et d'une tâche composée. Cette classe contient :

action L'action utilisée pour attribuer les ressources.

permissions Les permissions afin de savoir si une ressource a le droit ou pas d'accéder à la tâche.

timeout Le temps d'attente pour une ressource pour effectuer une tâche. Si le temps est écoulé, le moteur de workflow va assigner la tâche à une autre ressource. S'il n'y a pas de valeur, on attend à l'infini.

Resource

La classe abstraite resource représente une ressource du moteur de workflow. Elle ne sera pas présente en tant que telle dans le moteur de workflow mais est la généralisation d'une ressource. Cette classe contient :

idResource L'identifiant de la ressource dans la société.

User

La classe User représente un ressource du moteur de workflow qui est un employé de l'entreprise. Cette classe hérite de tous les attributs de la classe abstraite Ressource et ne contient pas d'autres attributs. Ce sera donc une instanciation concrète de la classe abstraite.

Group

La classe Group représente un ressource du moteur de workflow qui est un groupe d'employés de l'entreprise. Cette classe hérite de tous les attributs de la classe abstraite Ressource et ne contient pas d'autres attributs. Ce sera donc une instanciation concrète de la classe abstraite.

Action

La classe abstraite Action représente les différentes règles pour offrir ou assigner une tâche à une ressource. Elle ne sera pas présente en tant que telle dans le moteur de workflow mais est la généralisation d'une action. Cette classe contient :

id L'identifiant de l'action.

param Les paramètres pour effectuer l'action.

Il y aura pour chaque action possible une classe concrète qui l'implémente. Dans le diagramme la classes R-DA implémente l'action *Direct Allocation* et la classe R-RBA implémente l'action *Role-Based Allocation*. Chaque classe doit implémenter la méthode *run* qui est la méthode appelée par la tâche pour lancer l'action afin d'attribuer une ressource à celle-ci.

WorkList

La classe workList représente la liste de travail d'une ressource du moteur de workflow. Cette classe contient :

idResource La ressource.

idTask La tâche.

status Le statut de la tâche pour la ressource. Ce statut peut être un parmi les suivants : *created*, *offered*, *allocated*, *started*, *suspended*.

Log

La classe log représente l'historique de toutes les affectations d'une ressource à une tâche. Ceci va permettre au moteur de workflow de pouvoir choisir la bonne personne en fonction de l'historique. Par exemple, pour la règle *History-based Allocation* la tâche doit être offerte ou assignée à la ressource qui a déjà effectué le plus souvent cette tâche. Le moteur de workflow consultera le *log* afin de connaître la ressource appropriée pour ce travail. Cette classe contient :

idResource La ressource.

idTask La tâche.

date La date de clôture de la tâche.

7.6 Implémentation

Pour cette perspective, l'implémentation est assez directe. Tout au long de l'analyse, nous avons remarqué que chaque pattern peut être implémenté assez facilement. Soit en changeant le fichier XML d'entrée, soit en implémentant la logique de ce pattern dans notre moteur de workflow.

C'est pourquoi, nous pensons que pour cette perspective nous n'avons pas besoin des langages de coordination. Ceux-ci n'apportent pas une valeur ajoutée et la perspective en elle-même ne nécessite pas de besoins particuliers en terme d'expressivité.

Nous implémeterons donc cette perspective en Java. Nous ne rentrerons pas dans les détails dans le sens où cette implémentation sort du cadre de ce mémoire.

7.7 Conclusion

Nous venons d'effectuer une analyse de la perspective de ressources. Nous avons également donné le fil conducteur pour son implémentation. Cette étude est basée sur l'analyse des patterns de workflow [Russell *et al.*, 2004b]. Nous avons présenté des scénarios d'utilisation et aussi les différents problèmes liés à la persistance des ressources. Ensuite, nous avons complété le diagramme de domaine d'application et enfin nous avons abordé l'aspect d'implémentation.

Dans le chapitre suivant, nous analysons la perspective opérationnelle et présentons son implémentation.

Chapitre 8

Perspective opérationnelle

Dans ce chapitre, nous analysons la possibilité qu'une application extérieure exécute une ou plusieurs tâches du workflow. Comme présenté dans la perspective donnée, nous utilisons des web services pour la communication.

8.1 Description

Cette perspective permet à un acteur extérieur de lancer une tâche du workflow. Elle permet également au moteur de workflow de récupérer des données dans une application externe. Elle couvre donc tous les aspects d'interopérabilité entre notre moteur de workflow et les applications externes. Notons que comme la communication est bidirectionnelle, il est donc important de définir de part et d'autres comment s'effectuera cette communication.

Dans le reste de ce chapitre, nous allons présenter notre choix d'implémentation et les motivations qui justifient ce choix.

8.2 Implémentation

Dans cette perspective, on peut distinguer deux aspects :

1. l'appel d'une opération externe à partir de notre moteur de workflow (workflow vers environnement) ;
2. l'exécution d'une tâche du workflow à partir d'une application externe (environnement vers workflow).

Dans le premier cas, nous devons répondre aux questions suivantes :
- comment appeler cette opération externe ?
- comment lui passer des arguments ?
- comment récupérer la ou les valeurs de retour ?
- que faire en cas d'erreur lors de l'exécution de l'opération ?

Dans le second cas, nous devons fournir à l'appelant les réponses aux mêmes questions.

Actuellement, il existe de nombreuses solutions à cette problématique. On peut entre autre citer RPC (Remote Procedure Call), CORBA (Common Object Request Broker Architecture),

RMI (Remote Method Invocation) et les web services. Nous n'allons pas nous attarder sur les défauts et les qualités de chacune de ces méthodes. Pour notre part, nous avons choisi d'utiliser les web services [Alonso *et al.*, 2003].

Ceux-ci offrent un grand nombre d'avantages :

Interopérabilité : ils sont indépendants des plateformes et des langages de programmation ;

Facile à implémenter : le développement d'un web service dans un langage courant (Java, C#, Python, etc.) ne demande que peu d'efforts par rapport à RMI ou à CORBA.

Facile à déployer : le fait que tous les messages transitent en HTTP fait qu'un administrateur ne devra pas configurer de ports supplémentaires sur son firewall.

Évidemment, les web services ont aussi quelques défauts. On peut entre autre citer :

Format texte : ils utilisent un format texte. Ceci a donc un impact sur la performance car ceux-ci sont généralement moins performants qu'un format binaire.

Normes : toutes les normes ne sont pas encore établies.

Les nombreux avantages des web services font que nous les avons privilégiés aux autres technologies existantes.

Maintenant que nous avons choisi la technologie, on peut répondre effectivement aux nombreuses questions posées au début de cette section. Toutefois, nous allons conserver la distinction entre les deux aspects de l'interopérabilité.

8.3 Workflow vers environnement

Dans le cas où notre moteur de workflow doit appeler un web service, on doit définir un contrat qui liera notre moteur de workflow au web service. Ce contrat peut être assez simple et se limiter au fichier WSDL. Dans ce cas, le moteur de workflow connaît le nom de l'opération qu'il peut appeler ainsi que les différents paramètres de cette opération.

Il nous reste dès lors à répondre à une question qui est le comportement de notre moteur en cas d'erreur. Nous avons prévu dans ce cas là un comportement standard pour toutes les erreurs possibles. Celui-ci consiste tout simplement à ignorer l'erreur et à la recenser dans un fichier journal. Donc, si l'opération ne se termine pas avec succès ou si l'on reçoit une valeur de retour incorrecte, nous n'utilisons pas le résultat de l'opération et nous le signifions à l'utilisateur.

Ce traitement assez simpliste n'est évidemment pas viable dans le cadre d'une application professionnelle. Toutefois, un traitement des erreurs complet sort du cadre de ce mémoire.

8.4 Environnement vers workflow

Dans le cas inverse où c'est une application externe qui va déclencher un événement dans notre moteur de workflow, nous devons aussi définir un contrat. L'avantage est que dans ce cas, c'est nous qui avons le contrôle et on peut donc décrire précisément quelles sont les opérations permises par l'appelant.

Nous avons décidé de fournir trois web services :

`ws-list-tasks` : fournit la liste des tâches disponibles dans notre workflow pour une ressource. Il possède un argument en entrée qui est l'identifiant (unique) de la ressource et produit un argument en sortie. La valeur de l'argument de retour est un fichier XML qui contient le nom et l'identifiant de toutes les tâches disponibles.

`ws-get-task` : renvoie la description d'une tâche passée en paramètre. L'argument en entrée est l'identifiant (unique) d'une tâche. La valeur de retour est un fichier XML contenant les données en entrée et en sortie de la tâche.

`ws-set-task` : assigne les valeurs aux données de sortie d'une tâche passée en paramètre. Il y a deux arguments en entrée, le premier est l'identifiant (unique) d'une tâche et le second est le fichier XML qui contient toutes les valeurs de sortie de la tâche. Il n'y a pas de valeur de retour.

À l'aide de ces trois web services, on peut donc exécuter une tâche. Typiquement, une application externe utilisera ces web services dans l'ordre dans lesquels ils ont été présentés. Tout d'abord, l'application récupère la liste de tâches disponibles (`ws-list-tasks`) pour l'utilisateur courant. Ensuite, l'utilisateur sélectionne la tâche qu'il désire exécuter. L'application présente un écran permettant d'encoder les données de cette tâche (`ws-get-task`). Finalement, lorsque l'utilisateur a confirmé ces données, l'application soumet celles-ci au workflow (`ws-set-task`).

Remarquons qu'à nouveau nous avons simplifié la gestion des erreurs. Ainsi, si un utilisateur fournit par exemple l'identifiant d'une tâche qui n'existe pas dans le système, nous ne ferons rien d'autre que de signifier cette erreur dans un fichier journal. Toutes les autres erreurs seront traitées de façon identique (identifiant qui existe mais qui n'est pas disponible pour cette ressource, donnée n'appartenant pas à la tâche, etc.).

8.5 Conclusion

La perspective opérationnelle est certainement celle qui pour l'instant est encore la moins formalisée. Contrairement aux autres perspectives, il n'y a pas encore de patterns définis pour cette perspective. Nous n'avons trouvé que très peu de documentation sur cette perspective. La raison principale est selon nous que la portée de cette perspective est assez vaste. En effet, l'utilisation d'applications externes (via des web services ou autres) permet une extensibilité quasi infinie des moteurs de workflow. Dès lors, il est assez ardu de formaliser ces mécanismes sans introduire des limitations. Ceci explique donc que la présentation de cette perspective diffère des perspectives précédentes.

Chapitre 9

Exemple

Dans ce dernier chapitre, nous reprenons notre exemple initial de workflow et on synthétise au travers de celui-ci le travail effectué sur les quatre perspectives.

Pour rappel, ce workflow décrit la préparation d'un voyage (d'affaire ou touristique) dans une agence de voyages.

9.1 Modélisation

La figure 9.1 montre la modélisation graphique de notre workflow à l'aide des composants YAWL.

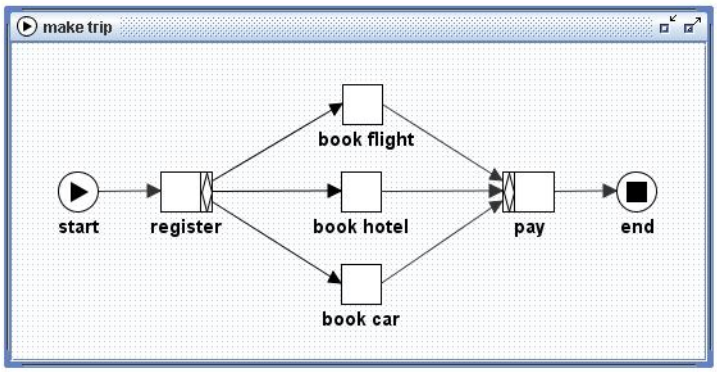

Figure 9.1 – Modélisation de l'exemple

9.2 Perspective de contrôle de flux

Le fichier de modélisation se trouve en annexe B.

9.3 Perspective de données

Le fichier de modélisation de la section précédente contient également les données.

9.4 Perspective de ressources

Nous définissons par tâches les ressources qui peuvent accomplir ce travail.

La tâche d'enregistrement du client est permisse à toutes les personnes de l'agence qui sont déclarées dans le groupe *g-agency*. L'identifiant *R-RA* indique qu'on assigne la tâche à un groupe de personnes qui est autorisé à exécuter la tâche Nous indiquons également un *timeout* d'une journée afin de pouvoir attribuer le travail à une autre ressource si la première n'a pas pu effectuer son travail.

```
<decomposition id="register" xsi:type="WebServiceGatewayFactsType">
    <resourceManager>
        <action>
            <id>R_RA</id>
            <parameter>g_agency</parameter>
        </action>
        <timeout unit="d">1</timeout>
    </resourceManager>
    [...]
</decomposition>
```

La tâche de réservation d'un avion ne peut être effectuée que par une personne de l'agence (*g-agency*) et qui a les droits *p-flight*. Nous indiquons également un *timeout* d'une journée.

```
<decomposition id="book_flight" xsi:type="WebServiceGatewayFactsType">
    <resourceManager>
        <action>
            <id>R_RA</id>
            <parameter>g_agency</parameter>
        </action>
        <timeout unit="d">1</timeout>
        <permission>p_flight</permission>
    </resourceManager>
    [...]
</decomposition>
```

Nous effectuons la même déclaration pour la tâche qui s'occupe de la réservation d'une voiture et d'un hôtel. La seule différence est la permission, pour réserver une voiture il faut avoir la permission *p-car* et pour réserver un hotel il faut avoir la permission *p-hotel*.

La tâche d'encaissement du voyage ne peut être effectuée que par une seule personne au sein de l'agence. Cette personne est le directeur de l'agence et son identifiant est *uid-123*. L'identifiant *R-DA* indique qu'on assigne la tâche à une personne bien précise.

```
<decomposition id="pay" xsi:type="WebServiceGatewayFactsType">
    <resourceManager>
        <action>
            <id>R_DA</id>
            <parameter>uid_123</parameter>
        </action>
    </resourceManager>
    [...]
</decomposition>
```

9.5 Perspective opérationnelle

Dans notre workflow, on peut par exemple utiliser un web service pour la réservation de l'hôtel. Nous devons donc avoir la définition du WSDL dans le fichier de modélisation. On appelera dès lors ce web service avec les bons paramètres dans la tâche « Book Hotel ».

Conclusion

La finalité de ce mémoire était d'étudier un moteur de workflow et d'examiner la possibilité d'implémenter ce moteur à l'aide des langages de coordination.

Après une étude approfondie de l'état de l'art, nous avons porté notre choix sur YAWL comme modèle de référence pour notre moteur de workflow. Nous avons aussi choisi le fichier XML de YAWL comme fichier de modélisation. Toutefois, lors de notre étude, nous nous sommes rendus compte qu'il était incomplet et nous l'avons donc complété.

L'un des résultats majeurs est sans aucun doute l'analyse et l'implémentation de la perspective de contrôle de flux. Celle-ci est au cœur de notre moteur de workflow. C'est pourquoi, nous avons fourni une analyse fort détaillée. De plus, pour l'implémentation, nous avons pu utiliser Reo. Reo est orienté flux et nous a fourni des connecteurs de base. Dès lors, cela a facilité grandement l'implémentation. Le résultat final montre que pour cette perspective, l'implémentation du moteur de workflow est plus aboutie que pour les autres perspectives.

Notons que nous ne sommes pas les seuls à nous être intéressés à YAWL. En effet, depuis quelques mois, il y a désormais un projet d'intégrer Reo dans YAWL[1].

Pour la perspective des données, nous avons aussi voulu utiliser un langage de coordination (Klava). Toutefois, nous nous sommes rendus compte que son utilisation aurait compliqué l'implémentation sans toutefois fournir de réels avantages. C'est pourquoi, nous avons préféré une solution plus proche du fichier de modélisation et plus directe.

Enfin, en ce qui concerne les deux dernières perspectives (ressources et opérationnelles), l'implémentation en utilisant des langages de coordination n'était, selon nous, pas pertinente. Néanmoins, leur étude nous a permis d'affiner l'analyse d'un moteur de workflow.

Les très nombreux patterns définis pour chaque perspective nous ont été très utiles. Ils offrent évidemment un cadre formel pour l'étude de chaque perspective. Ils nous ont permis aussi de garantir la correction de notre moteur de workflow. En effet, si on arrive à implémenter chaque pattern indépendemment dans notre moteur, on peut être sûr que celui-ci réalise une implémentation fidèle de chaque perspective.

La réalisation de ce mémoire n'a toutefois pas été exempte de difficultés. Nous avons eu en effet quelques soucis pour utiliser ReoLite efficacement. Son implémentation est incomplète et est peu documentée. D'autres langages, tels que MoCha, nous auraient peut-être posés moins de problèmes. Mais, ils auraient eu sans doute leur lot de désavantages. Pour MoCha, sa relative complexité (car beaucoup plus complet que Reo) aurait sans doute eu un grand impact sur notre temps d'apprentissage.

Avec un peu de recul, nous nous rendons compte que nous avons aussi ignoré certains aspects importants des moteurs de workflow. Nous n'avons pas tenu compte par exemple de la persistance des données. Nous avons aussi écarté les aspects de sécurité dans le workflow. La

[1] `http://db.cwi.nl/projecten/thema.php4?themanr=15&type=projects`

sécurité est bien évidemment un sujet inévitable dans un moteur de workflow. Toutefois, un traitement complet aurait nécessité un mémoire à part entière. De même, nous n'avons pas traité les patterns d'erreur. Ceux-ci sont transversaux à toutes les perspectives. Toutefois nous pensons que l'étendue du travail déjà réalisée a permis de présenter une vue d'ensemble des moteurs de workflow.

Ceci termine notre mémoire, nous espérons que celui-ci vous a donné un aperçu intéressant de l'implémentation d'un moteur de workflow et que sa lecture a été aussi enrichissante que sa rédaction l'a été pour nous.

Bibliographie

[Alonso *et al.*, 2003] Alonso, G., Casati, F., Kuno, H. et Machiraju, V. (2003). *Web Services - Concepts, Architectures and Applications*. Springer.

[Arbab et Talcott, 2002] Arbab, F. et Talcott, C. L., éditeurs (2002). *Coordination Models and Languages, 5th International Conference, COORDINATION 2002, YORK, UK, April 8-11, 2002, Proceedings*, volume 2315 de *Lecture Notes in Computer Science*. Springer.

[Bettini *et al.*, 2002] Bettini, L., Nicola, R. D. et Pugliese, R. (2002). Klava : a java package for distributed and mobile applications. *Softw. Pract. Exper.*, 32(14):1365–1394.

[Carriero et Gelernter, 1989] Carriero, N. et Gelernter, D. (1989). Linda in context. *Commun. ACM*, 32(4):444–458.

[Gamma *et al.*, 1995] Gamma, E., Helm, R., Johnson, R. et Vlissides, J. (1995). *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts.

[Hollingsworth, 1995] Hollingsworth, D. (1995). The workflow reference model. Rapport technique, Workflow Management Coalition.

[Kan, 2005] Kan, H. K. (2005). Design and implementation of an editor and simulators for constraint automata in the context of reo. SEN-E0512.

[Levan, 1999] Levan, S. K. (1999). *Le projet Workflow*. Eyrolles.

[Malone et Crowston, 1994] Malone, T. W. et Crowston, K. (1994). The interdisciplinary study of coordination. *ACM Comput. Surv.*, 26(1):87–119.

[Persson et Stirna, 2004] Persson, A. et Stirna, J., éditeurs (2004). *Advanced Information Systems Engineering, 16th International Conference, CAiSE 2004, Riga, Latvia, June 7-11, 2004, Proceedings*, volume 3084 de *Lecture Notes in Computer Science*. Springer.

[Russell *et al.*, 2004a] Russell, N., Arthur, Edmond, D. et van der Aalst, W. M. P. (2004a). Workflow resource patterns. *BETA Working Paper Series WP 127*.

[Russell *et al.*, 2004b] Russell, N., Arthur, Edmond, D. et van der Aalst, W. M. P. (2004b). Workflow resource patterns. *BETA Working Paper Series WP 127*.

[Russell *et al.*, 2006] Russell, N., Arthur, van der Aalst, W. M. P. et Mulyar, N. (2006). Workflow control-flow patterns : A revised view. *Distributed and Parallel Databases*.

[Singh, 1992] Singh, B. (1992). Interconnected roles (ir) : A coordinated model. technical. Rapport technique, Microelectronics and Computer Technology Corp.

[van der Aalst et Hofstede, 2002] van der Aalst, W. et Hofstede, A. (2002). Yawl : Yet another workflow language.

[van der Aalst *et al.*, 2003] van der Aalst, W. M. P., ter Hofstede, A. H. M., Kiepuszewski, B. et Barros, A. P. (2003). Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51.

[Wegner, 1997] Wegner, P. (1997). Why interaction is more powerful than algorithms. *Commun. ACM*, 40(5):80–91.

Glossaire

Réseau de Petri

Un réseau de Petri[1] est un modèle mathématique servant à représenter divers systèmes (informatiques, industriels, ...) travaillant sur des variables discrètes.

RUP - Rational Unified Process

RUP[2] est une méthode de prise en charge du cycle de vie d'un logiciel et donc du développement, pour les logiciels orientés objets. C'est une méthode générique, itérative et incrémentale.

SOAP - Simple Object Access Protocol

Simple Object Access Protocol[3] est un protocole de RPC orienté objet bâti sur XML.

Il permet la transmission de messages entre objets distants, ce qui veut dire qu'il autorise un objet à invoquer des méthodes d'objets physiquement situés sur un autre serveur. Le transfert se fait le plus souvent à l'aide du protocole HTTP, mais peut également se faire par un autre protocole, comme SMTP.

Le protocole SOAP est composé de deux parties :

- une enveloppe, contenant des informations sur le message lui-même afin de permettre son acheminement et son traitement,
- un modèle de données, définissant le format du message, c'est-à-dire les informations à transmettre.

UML - Unified Modeling Language

Unified Modeling Language[4] un langage graphique de modélisation des données et des traitements. C'est une formalisation très aboutie et non-propriétaire de la modélisation objet utilisée en génie logiciel. L'OMG travaille actuellement sur la version UML 2.1.

XML Query ou XQuery

XML Query[5] est un langage de requête permettant donc d'extraire des informations d'un document XML.

XML Query est une spécification du W3C. Sémantiquement proche de SQL, XML Query utilise la syntaxe XPath pour adresser des parties spécifiques d'un document XML.

[1] Site wikipedia, `http://fr.wikipedia.org/wiki/R%C3%A9seau_de_Petri` (visité le 20/08/2007).
[2] Site wikipedia, `http://fr.wikipedia.org/wiki/Unified_Process` (visité le 20/08/2007).
[3] Site wikipedia, `http://fr.wikipedia.org/wiki/SOAP` (visité le 20/08/2007).
[4] Site wikipedia, `http://fr.wikipedia.org/wiki/Unified_Modeling_Language` (visité le 20/08/2007).
[5] Site wikipedia, `http://fr.wikipedia.org/wiki/XML_Query` (visité le 20/08/2007).

XPath

XPath[6] est une syntaxe (non XML) pour désigner une portion d'un document XML. Initialement créé pour fournir une syntaxe et une sémantique aux fonctions communes à XPointer et XSL, XPath a rapidement été adopté par les développeurs comme un petit langage d'interrogation.

XSD - XML Shema

Un XSD ou un XML shema[7] est un langage de description de format de document XML permettant de définir la structure d'un document XML. La connaissance de la structure d'un document XML permet notamment de vérifier la validité de ce document. Un fichier de description de structure est donc lui-même un document XML.

Web service

Un web service [8] est un programme informatique permettant la communication et l'échange de données entre applications et systèmes hétérogènes dans des environnements distribués. Il s'agit donc d'un ensemble de fonctionnalités exposées sur Internet ou sur un Intranet, par et pour des applications ou machines, sans intervention humaine, et en temps réel.

WSDL - Web Service Description Language

Web Service Description Language[9] décrit une Interface publique d'accès à un web service, notamment dans le cadre d'architectures de type SOA (Service Oriented Architecture).

C'est une description basée sur le XML qui indique « comment communiquer pour utiliser le service » ; le Protocole de communication, et le format de messages requis pour communiquer avec ce service. Les opérations possibles et messages sont décrits de façon abstraite mais cette description renvoie à des protocoles réseaux et formats de messages concrets.

[6]Site wikipedia, `http://fr.wikipedia.org/wiki/XPath` (visité le 20/08/2007).

[7]Site wikipedia, `http://fr.wikipedia.org/wiki/XML_Schema` (visité le 20/08/2007).

[8]Site wikipedia, `http://fr.wikipedia.org/wiki/Web_service` (visité le 20/08/2007).

[9]Site wikipedia, `http://fr.wikipedia.org/wiki/WSDL` (visité le 20/08/2007).

Annexe A

Schéma XML du fichier de modélisation

Voici le schéma XML du fichier de modélisation avant les modifications.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XML Spy v4.4 U (http://www.xmlspy.com) by Lachlan Aldred (
    Queensland University of Technology) -->
<!--
<xs:schema targetNamespace="http://www.citi.qut.edu.au/yawl" xmlns:yawl="http://www.
    citi.qut.edu.au/yawl" xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified" attributeFormDefault="unqualified">
-->
<xs:schema targetNamespace="http://www.citi.qut.edu.au/yawl" xmlns:yawl="http://www.
    citi.qut.edu.au/yawl" xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified" attributeFormDefault="unqualified">
  <!--
  ##########################################################################################

    DECLARE ROOT ELEMENT - YAWL_Specification
  ##########################################################################################

  -->
  <xs:element name="specificationSet" type="yawl:SpecificationSetFactsType">
    <xs:unique name="SpecificationUnique">
      <xs:selector xpath="specification"/>
      <xs:field xpath="@uri"/>
    </xs:unique>
    <xs:key name="DecompositionKey">
      <xs:selector xpath="yawl:specification/yawl:decomposition"/>
      <xs:field xpath="@id"/>
    </xs:key>
    <xs:key name="NetElementKey">
      <xs:selector xpath="yawl:specification/*/yawl:processControlElements/*"/>
      <xs:field xpath="@id"/>
    </xs:key>
    <xs:keyref name="FlowsIntoForeignKey" refer="yawl:NetElementKey">
```

```
    <xs:selector xpath="yawl:specification/*/yawl:processControlElements/*/
        yawl:flowsInto/nextElementRef"/>
    <xs:field xpath="@id"/>
  </xs:keyref>
  <xs:keyref name="RemovesTokensForeignKey" refer="yawl:NetElementKey">
    <xs:selector xpath="yawl:specification/*/yawl:processControlElements/*/
        yawl:removesTokens"/>
    <xs:field xpath="@id"/>
  </xs:keyref>
  <xs:keyref name="DecomposeToForeignKey" refer="yawl:DecompositionKey">
    <xs:selector xpath="yawl:specification/*/yawl:processControlElements/*/
        yawl:decomposesTo"/>
    <xs:field xpath="@id"/>
  </xs:keyref>
</xs:element>
<!--
##############################################################################################

SIMPLE TYPES FROM VALUE TYPES
##############################################################################################

-->
<xs:simpleType name="ControlTypeCodeType">
  <xs:annotation>
    <xs:documentation>Encapsulates the range of relation T-->{AND, OR, XOR}</
        xs:documentation>
  </xs:annotation>
  <xs:restriction base="xs:string">
    <xs:enumeration value="and"/>
    <xs:enumeration value="or"/>
    <xs:enumeration value="xor"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="CreationModeCodeType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="static"/>
    <xs:enumeration value="dynamic"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="DecompositionIDType">
  <xs:restriction base="xs:string"/>
</xs:simpleType>
<xs:simpleType name="DocumentationType">
  <xs:restriction base="xs:string"/>
</xs:simpleType>
<xs:simpleType name="LabelType">
  <xs:restriction base="xs:string"/>
</xs:simpleType>
<xs:simpleType name="DirectionModeType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="input"/>
    <xs:enumeration value="output"/>
    <xs:enumeration value="both"/>
```

```
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="NameType">
  <xs:restriction base="xs:string">
    <xs:minLength value="1"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="NCNameType">
  <xs:restriction base="xs:NCName"/>
</xs:simpleType>
<xs:simpleType name="NetElementIDType">
  <xs:restriction base="xs:NMTOKEN"/>
</xs:simpleType>
<xs:simpleType name="NullType">
  <xs:restriction base="xs:string">
    <xs:maxLength value="0"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="PositiveIntegerType">
  <xs:restriction base="xs:positiveInteger"/>
</xs:simpleType>
<xs:simpleType name="URIType">
  <xs:restriction base="xs:anyURI"/>
</xs:simpleType>
<xs:simpleType name="VariableNameType">
  <xs:restriction base="xs:NMTOKEN"/>
</xs:simpleType>
<xs:simpleType name="XPathPredicateType">
  <xs:restriction base="xs:string">
    <xs:minLength value="1"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="XQueryType">
  <xs:restriction base="xs:string">
    <xs:minLength value="1"/>
  </xs:restriction>
</xs:simpleType>
<!--
##########################################################################################

COMPLEX TYPES FROM ENTITY TYPES
##########################################################################################

-->
<xs:complexType name="ControlTypeType">
  <xs:attribute name="code" type="yawl:ControlTypeCodeType" use="required"/>
</xs:complexType>
<xs:complexType name="CreationModeType">
  <xs:attribute name="code" type="yawl:CreationModeCodeType" use="required"/>
</xs:complexType>
<xs:complexType name="CustomSchemaNamespaceMappingType">
  <xs:attribute name="ncname" type="yawl:NCNameType" use="required"/>
</xs:complexType>
```

```
<xs:complexType name="DecompositionType">
  <xs:attribute name="id" type="yawl:DecompositionIDType" use="required"/>
</xs:complexType>
<xs:complexType name="DirectionType">
  <xs:attribute name="mode" type="yawl:DirectionModeType" use="required"/>
</xs:complexType>
<xs:complexType name="ExpressionType">
  <xs:attribute name="query" type="yawl:XQueryType" use="required"/>
</xs:complexType>
<xs:complexType name="ExternalNetElementType">
  <xs:attribute name="id" type="yawl:NetElementIDType" use="required"/>
</xs:complexType>
<xs:complexType name="FlowsIntoType">
  <xs:sequence>
    <xs:element name="nextElementRef" type="yawl:ExternalNetElementType"/>
    <xs:element name="predicate" type="yawl:PredicateType" minOccurs="0"/>
    <xs:element name="isDefaultFlow" type="yawl:NullType" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="LocationType">
  <xs:attribute name="uri" type="yawl:URIType"/>
</xs:complexType>
<xs:complexType name="NamespacePrefixType">
  <xs:attribute name="ncname" type="yawl:NCNameType" use="required"/>
</xs:complexType>
<xs:complexType name="NamespaceURI">
  <xs:attribute name="uri" type="yawl:URIType" use="required"/>
</xs:complexType>
<xs:complexType name="PredicateType">
  <xs:simpleContent>
    <xs:extension base="yawl:XPathPredicateType">
      <xs:attribute name="ordering" type="xs:integer"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
<xs:complexType name="TypeNameType">
  <xs:attribute name="ncname" type="yawl:NCNameType"/>
</xs:complexType>
<xs:complexType name="TypeType">
  <xs:sequence>
    <xs:element name="namespace" type="yawl:NamespacePrefixType"/>
    <xs:element name="typeName" type="yawl:TypeNameType"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="VariableType">
  <xs:attribute name="name" type="yawl:VariableNameType" use="required"/>
</xs:complexType>
<xs:complexType name="SpecificationType">
  <xs:attribute name="uri" type="yawl:URIType" use="required"/>
</xs:complexType>
<xs:complexType name="SpecificationSetType">
  <!--xs:attribute name="uri" type="yawl:URIType" use="required"/-->
</xs:complexType>
```

```
<!--
#############################################################################################

COMPLEX TYPES FROM FACT TYPE GROUPINGS
#############################################################################################

-->
<!--
#### TYPE #### CustomSchemaNamespaceMappingFactsType
-->
<xs:complexType name="CustomSchemaNamespaceMappingFactsType">
  <xs:complexContent>
    <xs:extension base="yawl:CustomSchemaNamespaceMappingType">
      <xs:sequence>
        <xs:element name="expandsTo" type="yawl:URIType"/>
        <xs:element name="definedAt" type="yawl:URIType"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<!--
#### TYPE #### DecompositionFactsType
-->
<xs:complexType name="DecompositionFactsType" abstract="true">
  <xs:complexContent>
    <xs:extension base="yawl:DecompositionType">
      <xs:sequence>
        <xs:element name="name" type="yawl:NameType" minOccurs="0"/>
        <xs:element name="documentation" type="yawl:DocumentationType" minOccurs="
            0"/>
        <xs:element name="inputParam" type="yawl:InputParameterFactsType"
            minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="outputExpression" type="yawl:ExpressionType" minOccurs="
            0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<!--
#### TYPE #### ExternalConditionFactsType
-->
<xs:complexType name="ExternalConditionFactsType">
  <xs:complexContent>
    <xs:extension base="yawl:ExternalNetElementFactsType"/>
  </xs:complexContent>
</xs:complexType>
<!--
#### TYPE #### ExternalNetElementFactsType
-->
<xs:complexType name="ExternalNetElementFactsType">
  <xs:complexContent>
    <xs:extension base="yawl:ExternalNetElementType">
      <xs:sequence>
```

```
        <xs:element name="name" minOccurs="0"/>
        <xs:element name="documentation" minOccurs="0"/>
        <xs:element name="flowsInto" type="yawl:FlowsIntoType" maxOccurs="
            unbounded"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<!--
#### TYPE #### ExternalTaskFactsType extends ExternalNetElementFactsType
-->
<xs:complexType name="ExternalTaskFactsType">
  <xs:complexContent>
    <xs:extension base="yawl:ExternalNetElementFactsType">
      <xs:sequence>
        <xs:element name="join" type="yawl:ControlTypeType"/>
        <xs:element name="split" type="yawl:ControlTypeType"/>
        <xs:element name="removesTokens" type="yawl:ExternalNetElementType"
            minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="removesTokensFromFlow" type="
            yawl:RemovesTokensFromFlowType" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="startingMappings" type="yawl:VarMappingSetType"
            minOccurs="0">
          <xs:key name="StartMappingExpressionKey">
            <xs:selector xpath="yawl:mapping/yawl:expression"/>
            <xs:field xpath="@query"/>
          </xs:key>
          <xs:key name="StartMappingMapToKey">
            <xs:selector xpath="yawl:mapping"/>
            <xs:field xpath="yawl:mapsTo"/>
          </xs:key>
        </xs:element>
        <xs:element name="completedMappings" type="yawl:VarMappingSetType"
            minOccurs="0">
          <xs:key name="CompletedMappingExpressionKey">
            <xs:selector xpath="yawl:mapping/yawl:expression"/>
            <xs:field xpath="@query"/>
          </xs:key>
          <xs:key name="CompletedMappingMapToKey">
            <xs:selector xpath="yawl:mapping"/>
            <xs:field xpath="yawl:mapsTo"/>
          </xs:key>
        </xs:element>
        <xs:element name="decomposesTo" type="yawl:DecompositionType" minOccurs="0
            "/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<!--
#### TYPE #### MultipleInstanceExternalTaskFactsType
-->
<xs:complexType name="MultipleInstanceExternalTaskFactsType">
```

```
  <xs:complexContent>
    <xs:extension base="yawl:ExternalTaskFactsType">
      <xs:sequence>
        <xs:element name="minimum" type="yawl:XQueryType"/>
        <xs:element name="maximum" type="yawl:XQueryType"/>
        <xs:element name="threshold" type="yawl:XQueryType"/>
        <xs:element name="creationMode" type="yawl:CreationModeType"/>
        <xs:element name="miDataInput">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="expression" type="yawl:ExpressionType"/>
              <xs:element name="splittingExpression" type="yawl:ExpressionType"/>
              <xs:element name="formalInputParam" type="yawl:VariableNameType"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="miDataOutput" minOccurs="0">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="formalOutputExpression" type="yawl:ExpressionType"
                  />
              <xs:element name="outputJoiningExpression" type="yawl:ExpressionType
                  "/>
              <xs:element name="resultAppliedToLocalVariable" type="
                  yawl:VariableNameType"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<!--
#### TYPE #### NetFactsType
-->
<xs:complexType name="NetFactsType">
  <xs:complexContent>
    <xs:extension base="yawl:DecompositionFactsType">
      <xs:sequence>
        <xs:element name="localVariable" type="yawl:VariableFactsType" minOccurs="
            0" maxOccurs="unbounded"/>
        <xs:element name="processControlElements">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="inputCondition" type="
                  yawl:ExternalConditionFactsType"/>
              <xs:choice maxOccurs="unbounded">
                <xs:element name="task" type="yawl:ExternalTaskFactsType"
                    minOccurs="0" maxOccurs="unbounded"/>
                <xs:element name="condition" type="yawl:ExternalConditionFactsType
                    " minOccurs="0" maxOccurs="unbounded"/>
              </xs:choice>
```

```
              <xs:element name="outputCondition" type="
                  yawl:OutputConditionFactsType"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<!--
#### TYPE #### OutputConditionFactsType extends ExternalNetElementType
-->
<xs:complexType name="OutputConditionFactsType">
  <xs:complexContent>
    <xs:restriction base="yawl:ExternalNetElementType">
      <xs:sequence>
        <xs:element name="name" minOccurs="0"/>
        <xs:element name="documentation" minOccurs="0"/>
      </xs:sequence>
      <xs:attribute name="id" type="yawl:NetElementIDType" use="required"/>
    </xs:restriction>
  </xs:complexContent>
</xs:complexType>
<!--
#### TYPE #### RemovesTokensFromFlowType
-->
<xs:complexType name="RemovesTokensFromFlowType">
  <xs:sequence>
    <xs:element name="flowSource" type="yawl:ExternalNetElementType"/>
    <xs:element name="flowDestination" type="yawl:ExternalNetElementType"/>
  </xs:sequence>
</xs:complexType>
<!--
#### TYPE #### VarMappingAndTransformType
-->
<xs:complexType name="VarMappingSetType">
  <xs:sequence>
    <xs:element name="mapping" type="yawl:VarMappingFactsType" maxOccurs="
        unbounded"/>
  </xs:sequence>
</xs:complexType>
<!--
#### TYPE #### VarMappingFactsType
-->
<xs:complexType name="VarMappingFactsType">
  <xs:sequence>
    <xs:element name="expression" type="yawl:ExpressionType"/>
    <xs:element name="mapsTo" type="yawl:VariableNameType"/>
  </xs:sequence>
</xs:complexType>
<!--
#### TYPE #### WebServiceGatewayFactsType
-->
```

```
<xs:complexType name="WebServiceGatewayFactsType">
  <xs:complexContent>
    <xs:extension base="yawl:DecompositionFactsType">
      <xs:sequence>
        <xs:element name="outputParam" type="yawl:OutputParamaterFactsType"
            minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="yawlService" minOccurs="0">
          <xs:complexType>
            <xs:choice>
              <xs:sequence>
                <xs:element name="wsdlLocation" type="xs:anyURI"/>
                <xs:element name="operationName" type="xs:NMTOKEN"/>
              </xs:sequence>
              <xs:sequence/>
            </xs:choice>
            <xs:attribute name="id" type="yawl:YAWLServiceIDType" use="required"/>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<!--
#### TYPE #### YAWLServiceType
-->
<xs:simpleType name="YAWLServiceIDType">
  <xs:restriction base="yawl:URIType"/>
</xs:simpleType>
<!--
#### TYPE #### YAWLSpecificationFactsType
-->
<xs:complexType name="YAWLSpecificationFactsType" final="#all">
  <xs:complexContent>
    <xs:extension base="yawl:SpecificationType">
      <xs:sequence>
        <xs:element name="name" type="yawl:NameType" minOccurs="0"/>
        <xs:element name="documentation" type="yawl:DocumentationType" minOccurs="
            0"/>
        <xs:element name="metaData" type="yawl:MetaDataType"/>
        <xs:element name="import" type="yawl:ImportType" minOccurs="0"/>
        <xs:any namespace="http://www.w3.org/2001/XMLSchema" processContents="lax"
             minOccurs="0"/>
        <xs:element name="rootNet" type="yawl:NetFactsType">
          <xs:unique name="VariableUnique">
            <xs:selector xpath="*"/>
            <xs:field xpath="@name"/>
          </xs:unique>
          <xs:unique name="OutputExpresionUnique">
            <xs:selector xpath="yawl:outputExpression"/>
            <xs:field xpath="@query"/>
          </xs:unique>
          <xs:key name="NetElementKey_Inner1">
            <xs:selector xpath="yawl:processControlElements/*"/>
```

```
      <xs:field xpath="@id"/>
    </xs:key>
    <xs:keyref name="RemovesTokensForeignKey1" refer="
        yawl:NetElementKey_Inner1">
      <xs:selector xpath="yawl:processControlElements/*/yawl:removesTokens"/
          >
      <xs:field xpath="@id"/>
    </xs:keyref>
    <xs:keyref name="RemovesTokensFromFlowForeignKey1" refer="
        yawl:NetElementKey_Inner1">
      <xs:selector xpath="yawl:processControlElements/*/
          yawl:removesTokensFromFlow/yawl:flowSource"/>
      <xs:field xpath="@id"/>
    </xs:keyref>
    <xs:keyref name="RemovesTokensFromFlowForeignKey2" refer="
        yawl:NetElementKey_Inner1">
      <xs:selector xpath="yawl:processControlElements/*/
          yawl:removesTokensFromFlow/yawl:flowDestination"/>
      <xs:field xpath="@id"/>
    </xs:keyref>
    <xs:keyref name="Flows_ToInnerKeyRef1" refer="yawl:NetElementKey_Inner1"
        >
      <xs:selector xpath="yawl:processControlElements/*/yawl:flowsInto/
          yawl:nextElementRef"/>
      <xs:field xpath="@id"/>
    </xs:keyref>
  </xs:element>
  <xs:element name="decomposition" type="yawl:DecompositionType" maxOccurs="
      unbounded">
    <xs:unique name="VariableUnique2">
      <xs:selector xpath="yawl:inputParam"/>
      <xs:field xpath="@name"/>
    </xs:unique>
    <xs:unique name="OutputExpresionUnique2">
      <xs:selector xpath="yawl:outputExpression"/>
      <xs:field xpath="@query"/>
    </xs:unique>
    <xs:unique name="OutputParamUnique">
      <xs:selector xpath="yawl:outputParam"/>
      <xs:field xpath="@name"/>
    </xs:unique>
    <xs:key name="NetElementKey_Inner2">
      <xs:selector xpath="yawl:processControlElements/*"/>
      <xs:field xpath="@id"/>
    </xs:key>
    <xs:keyref name="Flows_ToInnerKeyRef2" refer="yawl:NetElementKey_Inner2"
        >
      <xs:selector xpath="yawl:processControlElements/*/yawl:flowsInto/
          yawl:nextElementRef"/>
      <xs:field xpath="@id"/>
    </xs:keyref>
    <xs:keyref name="RemovesTokensForeignKey2" refer="
        yawl:NetElementKey_Inner2">
```

```
            <xs:selector xpath="yawl:processControlElements/*/yawl:removesTokens"/
                >
            <xs:field xpath="@id"/>
          </xs:keyref>
          <xs:keyref name="RemovesTokensFromFlowForeignKey3" refer="
              yawl:NetElementKey_Inner2">
            <xs:selector xpath="yawl:processControlElements/*/
                yawl:removesTokensFromFlow/yawl:flowSource"/>
            <xs:field xpath="@id"/>
          </xs:keyref>
          <xs:keyref name="RemovesTokensFromFlowForeignKey4" refer="
              yawl:NetElementKey_Inner2">
            <xs:selector xpath="yawl:processControlElements/*/
                yawl:removesTokensFromFlow/yawl:flowDestination"/>
            <xs:field xpath="@id"/>
          </xs:keyref>
          <xs:unique name="YawlServiceUnique">
            <xs:selector xpath="yawl:yawlService"/>
            <xs:field xpath="@id"/>
          </xs:unique>
          <!-- -->
        </xs:element>
        <xs:element name="importedNet" type="xs:anyURI" minOccurs="0" maxOccurs="
            unbounded"/>
        <!--xs:element name="webServiceGateways" type="WebServiceGatewaysType"
            maxOccurs="unbounded"/-->
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<xs:complexType name="SpecificationSetFactsType">
  <xs:complexContent>
    <xs:extension base="yawl:SpecificationSetType">
      <xs:sequence>
        <xs:element name="specification" type="yawl:YAWLSpecificationFactsType"
            maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<!--
################################################################################

BEGIN VARIABLE FACTS TYPES
################################################################################

-->
<!--
#### TYPE #### VariableFactsType
-->
<xs:complexType name="VariableFactsType">
  <xs:complexContent>
    <xs:extension base="yawl:VariableType">
```

```
      <xs:sequence>
        <xs:element name="type" type="xs:anyURI">
          <xs:annotation>
            <xs:documentation>The range of this attribute is restricted to any
                namespace delimited Schema type.  BTW if the Schema is a custom
                schema then it is highly advisable to use the SchemaLocation to
                Namespace binding to allow the engine to find, and use, the Schema
                .</xs:documentation>
          </xs:annotation>
        </xs:element>
        <xs:element name="initialValue" type="xs:string" minOccurs="0"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<!--
#### TYPE #### ParamaterFactsType
-->
<xs:complexType name="InputParameterFactsType">
  <xs:complexContent>
    <xs:extension base="yawl:VariableType">
      <xs:sequence>
        <xs:element name="type" type="xs:anyURI"/>
        <xs:choice minOccurs="0">
          <xs:element name="initialValue" type="xs:string" minOccurs="0"/>
          <xs:element name="mandatory" minOccurs="0"/>
        </xs:choice>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<!--
#### TYPE #### OutputParamaterFactsType
-->
<xs:complexType name="OutputParamaterFactsType">
  <xs:complexContent>
    <xs:restriction base="yawl:InputParameterFactsType">
      <xs:sequence>
        <xs:element name="type" type="xs:anyURI"/>
        <xs:element name="mandatory" minOccurs="0"/>
      </xs:sequence>
      <xs:attribute name="name" type="yawl:VariableNameType" use="required"/>
    </xs:restriction>
  </xs:complexContent>
</xs:complexType>
<!--
###############################################################################################

BEGIN Ancilliary TYPES
###############################################################################################

-->
<!--
```

```
#### TYPE #### MetaDataType
-->
<xs:complexType name="MetaDataType">
  <xs:annotation>
    <xs:documentation>Uses meta data specification of the dublin core: http://
        dublincore.org/usage/terms/dc/current-elements/</xs:documentation>
  </xs:annotation>
  <xs:sequence>
    <xs:element name="title" type="xs:normalizedString" minOccurs="0"/>
    <xs:element name="creator" type="yawl:IndividualOrOrganisationType" minOccurs=
        "0" maxOccurs="unbounded"/>
    <xs:element name="subject" type="xs:string" minOccurs="0" maxOccurs="unbounded
        "/>
    <xs:element name="description" type="xs:normalizedString" minOccurs="0"/>
    <xs:element name="publisher" type="yawl:IndividualOrOrganisationType"
        minOccurs="0"/>
    <xs:element name="contributor" type="yawl:IndividualOrOrganisationType"
        minOccurs="0" maxOccurs="unbounded"/>
    <xs:element name="coverage" type="xs:string" minOccurs="0"/>
    <xs:element name="rights" type="xs:normalizedString" minOccurs="0"/>
    <xs:element name="replaces" type="xs:anyURI" minOccurs="0" maxOccurs="
        unbounded"/>
    <xs:element name="modification" minOccurs="0" maxOccurs="unbounded">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="by" type="yawl:IndividualOrOrganisationType"/>
          <xs:element name="on" type="xs:dateTime"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
    <xs:element name="validFrom" type="xs:date" minOccurs="0"/>
    <xs:element name="validUntil" type="xs:date" minOccurs="0"/>
    <xs:element name="created" type="xs:date" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>
<!--
#### TYPE #### IndividualOrOrganisationType
-->
<xs:complexType name="IndividualOrOrganisationType">
  <xs:sequence>
    <xs:element name="name" type="xs:string"/>
    <xs:choice maxOccurs="4">
      <xs:element name="address" type="yawl:AddressType"/>
      <xs:element name="emailAddress" type="xs:anyURI"/>
    </xs:choice>
    <xs:element name="description" type="xs:normalizedString" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>
<!--
#### TYPE #### AddressType
-->
<xs:complexType name="AddressType">
  <xs:sequence>
```

```
    <xs:choice>
      <xs:element name="numAndStreet">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="number">
              <xs:simpleType>
                <xs:restriction base="xs:string">
                  <xs:pattern value="\d*(/|-)*\d+"/>
                </xs:restriction>
              </xs:simpleType>
            </xs:element>
            <xs:element name="street">
              <xs:simpleType>
                <xs:restriction base="xs:string">
                  <xs:pattern value="\D+\s+[S|s|R|r|A|a|C|c|B|b|o](treet|oad|venue
                      |t|d|ve|rescent|lvd|ther)"/>
                </xs:restriction>
              </xs:simpleType>
            </xs:element>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="GPOBox" type="xs:positiveInteger"/>
    </xs:choice>
    <xs:element name="suburbOrTown" type="xs:string" minOccurs="0"/>
    <xs:element name="postOrZipCode">
      <xs:simpleType>
        <xs:restriction base="xs:positiveInteger">
          <xs:minInclusive value="1000"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:element>
    <xs:element name="state" type="xs:string" minOccurs="0"/>
    <xs:element name="country" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
<!-- #### TYPE ####  ImportType for importing Schmeas from other documents -->
<xs:complexType name="ImportType">
  <xs:attribute name="namespace" type="xs:anyURI"/>
  <xs:attribute name="location" type="xs:anyURI"/>
</xs:complexType>
<!-- #### TYPE #### SchemaType for declaring data types to use in YAWL - using XML
     Schema -->
<xs:complexType name="SchemaType">
  <xs:sequence>
    <xs:element name="type" maxOccurs="unbounded">
      <xs:complexType>
        <xs:sequence>
          <xs:any namespace="http://www.w3.org/2001/XMLSchema" processContents="
              lax" maxOccurs="unbounded"/>
        </xs:sequence>
        <xs:attribute name="targetNamespace" type="xs:anyURI"/>
      </xs:complexType>
```

```
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

Listing A.1 – Schéma XML du fichier de modélisation (avant modifications)

Voici le shéma XML du fichier de modélisation après les modifications.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XML Spy v4.4 U (http://www.xmlspy.com) by Lachlan Aldred (
    Queensland University of Technology) -->
<xs:schema targetNamespace="http://www.citi.qut.edu.au/yawl" elementFormDefault="
    qualified" attributeFormDefault="unqualified" xmlns:xs="http://www.w3.org/2001/
    XMLSchema" xmlns:yawl="http://www.citi.qut.edu.au/yawl">
  <!--
  ################################################################################

    DECLARE ROOT ELEMENT - YAWL_Specification
  ################################################################################

  -->
  <xs:element name="specificationSet" type="yawl:SpecificationSetFactsType">
    <xs:unique name="SpecificationUnique">
      <xs:selector xpath="specification"/>
      <xs:field xpath="@uri"/>
    </xs:unique>
  </xs:element>
  <!--
  ################################################################################

  SIMPLE TYPES FROM VALUE TYPES
  ################################################################################

  -->
  <xs:complexType name="ResourceManagerType">
    <xs:sequence>
      <xs:element name="action" type="yawl:ActionType" minOccurs="0"/>
      <xs:element name="permission" type="xs:string" minOccurs="0"/>
      <xs:element name="timeout" type="yawl:TimeoutType" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="ActionType">
    <xs:sequence>
      <xs:element name="id" type="xs:string"/>
      <xs:sequence>
        <xs:element name="parameter" type="xs:string" minOccurs="0"/>
      </xs:sequence>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="TimeoutType">
    <xs:complexContent>
      <xs:restriction base="xs:int">
          <xs:attribute name="unit" type="xs:string" use="required"/>
```

```
    </xs:restriction>
  </xs:complexContent>
</xs:complexType>

<xs:simpleType name="ControlTypeCodeType">
  <xs:annotation>
    <xs:documentation>Encapsulates the range of relation T——&gt;{AND, OR, XOR}</
        xs:documentation>
  </xs:annotation>
  <xs:restriction base="xs:string">
    <xs:enumeration value="and"/>
    <xs:enumeration value="or"/>
    <xs:enumeration value="xor"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="CreationModeCodeType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="static"/>
    <xs:enumeration value="dynamic"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="DecompositionIDType">
  <xs:restriction base="xs:NCName"/>
</xs:simpleType>
<xs:simpleType name="DocumentationType">
  <xs:restriction base="xs:string"/>
</xs:simpleType>
<xs:simpleType name="LabelType">
  <xs:restriction base="xs:string"/>
</xs:simpleType>
<xs:simpleType name="DirectionModeType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="input"/>
    <xs:enumeration value="output"/>
    <xs:enumeration value="both"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="NameType">
  <xs:restriction base="xs:string">
    <xs:minLength value="1"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="NCNameType">
  <xs:restriction base="xs:NCName"/>
</xs:simpleType>
<xs:simpleType name="NetElementIDType">
  <xs:restriction base="xs:NMTOKEN"/>
</xs:simpleType>
<xs:simpleType name="NullType">
  <xs:restriction base="xs:string">
    <xs:maxLength value="0"/>
  </xs:restriction>
</xs:simpleType>
```

```
<xs:simpleType name="PositiveIntegerType">
  <xs:restriction base="xs:positiveInteger"/>
</xs:simpleType>
<xs:simpleType name="URIType">
  <xs:restriction base="xs:anyURI"/>
</xs:simpleType>
<xs:simpleType name="VariableNameType">
  <xs:restriction base="xs:NMTOKEN"/>
</xs:simpleType>
<xs:simpleType name="XPathPredicateType">
  <xs:restriction base="xs:string">
    <xs:minLength value="1"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="XQueryType">
  <xs:restriction base="xs:string">
    <xs:minLength value="1"/>
  </xs:restriction>
</xs:simpleType>
<!--
##########################################################################################

COMPLEX TYPES FROM ENTITY TYPES
##########################################################################################

-->
<xs:complexType name="ControlTypeType">
  <xs:attribute name="code" type="yawl:ControlTypeCodeType" use="required"/>
</xs:complexType>
<xs:complexType name="CreationModeType">
  <xs:attribute name="code" type="yawl:CreationModeCodeType" use="required"/>
</xs:complexType>
<xs:complexType name="CustomSchemaNamespaceMappingType">
  <xs:attribute name="ncname" type="yawl:NCNameType" use="required"/>
</xs:complexType>
<xs:complexType name="DecompositionType">
  <xs:attribute name="id" type="yawl:DecompositionIDType" use="required"/>
</xs:complexType>
<xs:complexType name="DirectionType">
  <xs:attribute name="mode" type="yawl:DirectionModeType" use="required"/>
</xs:complexType>
<xs:complexType name="ExpressionType">
  <xs:attribute name="query" type="yawl:XQueryType" use="required"/>
</xs:complexType>
<xs:complexType name="ExternalNetElementType">
  <xs:attribute name="id" type="yawl:NetElementIDType" use="required"/>
</xs:complexType>
<xs:complexType name="FlowsIntoType">
  <xs:sequence>
    <xs:element name="nextElementRef" type="yawl:ExternalNetElementType"/>
    <xs:element name="predicate" type="yawl:PredicateType" minOccurs="0"/>
    <xs:element name="isDefaultFlow" type="yawl:NullType" minOccurs="0"/>
    <xs:element name="documentation" type="xs:string" minOccurs="0"/>
```

```
  </xs:sequence>
</xs:complexType>
<xs:complexType name="LocationType">
  <xs:attribute name="uri" type="yawl:URIType"/>
</xs:complexType>
<xs:complexType name="NamespacePrefixType">
  <xs:attribute name="ncname" type="yawl:NCNameType" use="required"/>
</xs:complexType>
<xs:complexType name="NamespaceURI">
  <xs:attribute name="uri" type="yawl:URIType" use="required"/>
</xs:complexType>
<xs:complexType name="PredicateType">
  <xs:simpleContent>
    <xs:extension base="yawl:XPathPredicateType">
      <xs:attribute name="ordering" type="xs:integer"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
<xs:complexType name="TypeNameType">
  <xs:attribute name="ncname" type="yawl:NCNameType"/>
</xs:complexType>
<xs:complexType name="TypeType">
  <xs:sequence>
    <xs:element name="namespace" type="yawl:NamespacePrefixType"/>
    <xs:element name="typeName" type="yawl:TypeNameType"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="SpecificationType">
  <xs:attribute name="uri" type="yawl:URIType" use="required"/>
</xs:complexType>
<xs:complexType name="SpecificationSetType">
  <!--xs:attribute name="uri" type="yawl:URIType" use="required"/-->
</xs:complexType>
<!--
################################################################################

COMPLEX TYPES FROM FACT TYPE GROUPINGS
################################################################################

-->
<!--
#### TYPE #### CustomSchemaNamespaceMappingFactsType
-->
<xs:complexType name="CustomSchemaNamespaceMappingFactsType">
  <xs:complexContent>
    <xs:extension base="yawl:CustomSchemaNamespaceMappingType">
      <xs:sequence>
        <xs:element name="expandsTo" type="yawl:URIType"/>
        <xs:element name="definedAt" type="yawl:URIType"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

```
<!--
#### TYPE #### DecompositionFactsType
-->
<xs:complexType name="DecompositionFactsType" abstract="true">
  <xs:complexContent>
    <xs:extension base="yawl:DecompositionType">
      <xs:sequence>
        <xs:element name="name" type="yawl:NameType" minOccurs="0"/>
        <xs:element name="documentation" type="yawl:DocumentationType" minOccurs="
            0"/>
        <xs:element name="inputParam" type="yawl:InputParameterFactsType"
            minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="outputParam" type="yawl:OutputParameterFactsType"
            minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="resourceManager" type="yawl:ResourceManagerType"
            minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<!--
#### TYPE #### ExternalConditionFactsType
-->
<xs:complexType name="ExternalConditionFactsType">
  <xs:complexContent>
    <xs:extension base="yawl:ExternalNetElementFactsType"/>
  </xs:complexContent>
</xs:complexType>
<!--
#### TYPE #### ExternalNetElementFactsType
-->
<xs:complexType name="ExternalNetElementFactsType">
  <xs:complexContent>
    <xs:extension base="yawl:ExternalNetElementType">
      <xs:sequence>
        <xs:element name="name" minOccurs="0"/>
        <xs:element name="documentation" minOccurs="0"/>
        <xs:element name="flowsInto" type="yawl:FlowsIntoType" maxOccurs="
            unbounded"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<!--
#### TYPE #### ExternalTaskFactsType extends ExternalNetElementFactsType
-->
<xs:complexType name="ExternalTaskFactsType">
  <xs:complexContent>
    <xs:extension base="yawl:ExternalNetElementFactsType">
      <xs:sequence>
        <xs:element name="join" type="yawl:ControlTypeType"/>
        <xs:element name="split" type="yawl:ControlTypeType"/>
```

```
        <xs:element name="removesTokens" type="yawl:ExternalNetElementType"
            minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="removesTokensFromFlow" type="
            yawl:RemovesTokensFromFlowType" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="startingMappings" type="yawl:VarMappingSetType"
            minOccurs="0">
          <xs:key name="StartMappingExpressionKey">
            <xs:selector xpath="yawl:mapping/yawl:expression"/>
            <xs:field xpath="@query"/>
          </xs:key>
          <xs:key name="StartMappingMapToKey">
            <xs:selector xpath="yawl:mapping"/>
            <xs:field xpath="yawl:mapsTo"/>
          </xs:key>
        </xs:element>
        <xs:element name="completedMappings" type="yawl:VarMappingSetType"
            minOccurs="0">
          <xs:key name="CompletedMappingExpressionKey">
            <xs:selector xpath="yawl:mapping/yawl:expression"/>
            <xs:field xpath="@query"/>
          </xs:key>
          <xs:key name="CompletedMappingMapToKey">
            <xs:selector xpath="yawl:mapping"/>
            <xs:field xpath="yawl:mapsTo"/>
          </xs:key>
        </xs:element>
        <xs:element name="enablementMappings" type="yawl:VarMappingSetType"
            minOccurs="0">
          <xs:key name="EnablementMappingExpressionKey">
            <xs:selector xpath="yawl:mapping/yawl:expression"/>
            <xs:field xpath="@query"/>
          </xs:key>
          <xs:key name="EnablementMappingMapToKey">
            <xs:selector xpath="yawl:mapping"/>
            <xs:field xpath="yawl:mapsTo"/>
          </xs:key>
        </xs:element>
        <xs:element name="decomposesTo" type="yawl:DecompositionType" minOccurs="0
            "/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<!--
#### TYPE #### MultipleInstanceExternalTaskFactsType
-->
<xs:complexType name="MultipleInstanceExternalTaskFactsType">
  <xs:complexContent>
    <xs:extension base="yawl:ExternalTaskFactsType">
      <xs:sequence>
        <xs:element name="minimum" type="yawl:XQueryType"/>
        <xs:element name="maximum" type="yawl:XQueryType"/>
        <xs:element name="threshold" type="yawl:XQueryType"/>
```

```
        <xs:element name="creationMode" type="yawl:CreationModeType"/>
        <xs:element name="miDataInput">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="expression" type="yawl:ExpressionType"/>
              <xs:element name="splittingExpression" type="yawl:ExpressionType"/>
              <xs:element name="formalInputParam" type="yawl:VariableNameType"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="miDataOutput" minOccurs="0">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="formalOutputExpression" type="yawl:ExpressionType"
                  />
              <xs:element name="outputJoiningExpression" type="yawl:ExpressionType
                  "/>
              <xs:element name="resultAppliedToLocalVariable" type="
                  yawl:VariableNameType"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<!--
#### TYPE #### NetFactsType
-->
<xs:complexType name="NetFactsType">
  <xs:complexContent>
    <xs:extension base="yawl:DecompositionFactsType">
      <xs:sequence>
        <xs:element name="localVariable" type="yawl:VariableFactsType" minOccurs="
            0" maxOccurs="unbounded"/>
        <xs:element name="processControlElements">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="inputCondition" type="
                  yawl:ExternalConditionFactsType"/>
              <xs:choice maxOccurs="unbounded">
                <xs:element name="task" type="yawl:ExternalTaskFactsType"
                    minOccurs="0" maxOccurs="unbounded"/>
                <xs:element name="condition" type="yawl:ExternalConditionFactsType
                    " minOccurs="0" maxOccurs="unbounded"/>
              </xs:choice>
              <xs:element name="outputCondition" type="
                  yawl:OutputConditionFactsType"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
      <xs:attribute name="isRootNet" type="xs:boolean"/>
```

```
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<!--
#### TYPE #### OutputConditionFactsType extends ExternalNetElementType
-->
<xs:complexType name="OutputConditionFactsType">
  <xs:complexContent>
    <xs:extension base="yawl:ExternalNetElementType">
      <xs:sequence>
        <xs:element name="name" minOccurs="0"/>
        <xs:element name="documentation" minOccurs="0"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<!--
#### TYPE #### RemovesTokensFromFlowType
-->
<xs:complexType name="RemovesTokensFromFlowType">
  <xs:sequence>
    <xs:element name="flowSource" type="yawl:ExternalNetElementType"/>
    <xs:element name="flowDestination" type="yawl:ExternalNetElementType"/>
  </xs:sequence>
</xs:complexType>
<!--
#### TYPE #### VarMappingAndTransformType
-->
<xs:complexType name="VarMappingSetType">
  <xs:sequence>
    <xs:element name="mapping" type="yawl:VarMappingFactsType" maxOccurs="
        unbounded"/>
  </xs:sequence>
</xs:complexType>
<!--
#### TYPE #### VarMappingFactsType
-->
<xs:complexType name="VarMappingFactsType">
  <xs:sequence>
    <xs:element name="expression" type="yawl:ExpressionType"/>
    <xs:element name="mapsTo" type="yawl:VariableNameType"/>
  </xs:sequence>
</xs:complexType>
<!--
#### TYPE #### WebServiceGatewayFactsType
-->
<xs:complexType name="WebServiceGatewayFactsType">
  <xs:complexContent>
    <xs:extension base="yawl:DecompositionFactsType">
      <xs:sequence>
        <xs:element name="enablementParam" type="yawl:InputParameterFactsType"
            minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="yawlService" minOccurs="0">
```

```
          <xs:complexType>
            <xs:choice>
              <xs:sequence>
                <xs:element name="wsdlLocation" type="xs:anyURI"/>
                <xs:element name="operationName" type="xs:NMTOKEN"/>
              </xs:sequence>
              <xs:sequence/>
            </xs:choice>
            <xs:attribute name="id" type="yawl:YAWLServiceIDType" use="required"/>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
      <xs:anyAttribute processContents="skip"/>
    </xs:extension>
    <!-- Extended Attributes (AJH) -->
  </xs:complexContent>
</xs:complexType>
<!--
#### TYPE #### YAWLServiceType
-->
<xs:simpleType name="YAWLServiceIDType">
  <xs:restriction base="yawl:URIType"/>
</xs:simpleType>
<!--
#### TYPE #### YAWLSpecificationFactsType
-->
<xs:complexType name="YAWLSpecificationFactsType" final="#all">
  <xs:complexContent>
    <xs:extension base="yawl:SpecificationType">
      <xs:sequence>
        <xs:element name="name" type="yawl:NameType" minOccurs="0"/>
        <xs:element name="documentation" type="yawl:DocumentationType" minOccurs="
            0"/>
        <xs:element name="metaData" type="yawl:MetaDataType"/>
        <xs:any namespace="http://www.w3.org/2001/XMLSchema" processContents="lax"
             minOccurs="0"/>
        <xs:element name="decomposition" type="yawl:DecompositionType" maxOccurs="
            unbounded">
          <xs:key name="DecompositionKey">
            <xs:selector xpath="yawl:decomposition"/>
            <xs:field xpath="@id"/>
          </xs:key>
          <xs:keyref name="DecomposeToForeignKey" refer="yawl:DecompositionKey">
            <xs:selector xpath="*/yawl:processControlElements/*/yawl:decomposesTo"
                />
            <xs:field xpath="@id"/>
          </xs:keyref>
          <xs:unique name="VariableUnique">
            <xs:selector xpath="yawl:inputParam | yawl:localVariable"/>
            <xs:field xpath="yawl:name"/>
          </xs:unique>
          <xs:unique name="OutputExpresionUnique">
            <xs:selector xpath="yawl:outputExpression"/>
```

```
            <xs:field xpath="@query"/>
          </xs:unique>
          <xs:unique name="OutputParamUnique">
            <xs:selector xpath="yawl:outputParam"/>
            <xs:field xpath="yawl:name"/>
          </xs:unique>
          <xs:unique name="EnablementParamUnique">
            <xs:selector xpath="yawl:enablementParam"/>
            <xs:field xpath="yawl:name"/>
          </xs:unique>
          <xs:key name="NetElementKey_Inner">
            <xs:selector xpath="yawl:processControlElements/*"/>
            <xs:field xpath="@id"/>
          </xs:key>
          <xs:keyref name="Flows_ToInnerKeyRef" refer="yawl:NetElementKey_Inner">
            <xs:selector xpath="yawl:processControlElements/*/yawl:flowsInto/
                yawl:nextElementRef"/>
            <xs:field xpath="@id"/>
          </xs:keyref>
          <xs:keyref name="RemovesTokensForeignKey" refer="
              yawl:NetElementKey_Inner">
            <xs:selector xpath="yawl:processControlElements/*/yawl:removesTokens"/
                >
            <xs:field xpath="@id"/>
          </xs:keyref>
          <xs:keyref name="RemovesTokensFromFlowForeignKey" refer="
              yawl:NetElementKey_Inner">
            <xs:selector xpath="yawl:processControlElements/*/
                yawl:removesTokensFromFlow/yawl:flowSource"/>
            <xs:field xpath="@id"/>
          </xs:keyref>
          <xs:keyref name="RemovesTokensFromFlowForeignKey2" refer="
              yawl:NetElementKey_Inner">
            <xs:selector xpath="yawl:processControlElements/*/
                yawl:removesTokensFromFlow/yawl:flowDestination"/>
            <xs:field xpath="@id"/>
          </xs:keyref>
          <xs:unique name="YawlServiceUnique">
            <xs:selector xpath="yawl:yawlService"/>
            <xs:field xpath="@id"/>
          </xs:unique>
          <!-- -->
        </xs:element>
        <xs:element name="importedNet" type="xs:anyURI" minOccurs="0" maxOccurs="
            unbounded"/>
        <!--xs:element name="webServiceGateways" type="WebServiceGatewaysType"
            maxOccurs="unbounded"/-->
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<xs:complexType name="SpecificationSetFactsType">
  <xs:complexContent>
```

```
    <xs:extension base="yawl:SpecificationSetType">
      <xs:sequence>
        <xs:element name="specification" type="yawl:YAWLSpecificationFactsType"
            maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="version" use="required">
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:enumeration value="Beta 7.1"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:attribute>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<!--
##########################################################################################

BEGIN VARIABLE FACTS TYPES
##########################################################################################

-->
<!--
#### TYPE #### VariableFactsType
-->
<xs:complexType name="VariableBaseType">
  <xs:sequence>
    <xs:element name="documentation" type="xs:string" minOccurs="0"/>
    <xs:choice>
      <xs:sequence>
        <xs:element name="name" type="xs:NCName"/>
        <xs:choice>
          <xs:sequence>
            <xs:element name="type" type="xs:NCName"/>
            <xs:element name="namespace" type="xs:anyURI" minOccurs="0"/>
          </xs:sequence>
          <xs:element name="isUntyped"/>
        </xs:choice>
      </xs:sequence>
      <xs:element name="element" type="xs:NCName"/>
    </xs:choice>
  </xs:sequence>
</xs:complexType>
<!--
#### TYPE #### VariableFactsType
-->
<xs:complexType name="VariableFactsType">
  <xs:complexContent>
    <xs:extension base="yawl:VariableBaseType">
      <xs:sequence>
        <xs:element name="initialValue" type="xs:string" minOccurs="0"/>
        <xs:element name="static" type="xs:boolean" minOccurs="0"/>
        <xs:element name="wsdl" type="xs:string" minOccurs="0"/>
```

```
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<!--
#### TYPE #### OutputParamaterFactsType
-->
<xs:complexType name="OutputParameterFactsType">
  <xs:complexContent>
    <xs:extension base="yawl:VariableBaseType">
      <xs:sequence>
        <xs:element name="mandatory" minOccurs="0"/>
        <xs:element name="isCutThroughParam" minOccurs="0"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<!--
#### TYPE #### ParamaterFactsType
-->
<xs:complexType name="InputParameterFactsType">
  <xs:complexContent>
    <xs:extension base="yawl:VariableBaseType">
      <xs:sequence>
        <xs:choice minOccurs="0">
          <xs:element name="initialValue" type="xs:string" minOccurs="0"/>
          <xs:element name="mandatory" minOccurs="0"/>
        </xs:choice>
      </xs:sequence>
      <xs:anyAttribute processContents="skip"/>
    </xs:extension>
    <!-- Extended Attributes (AJH) -->
  </xs:complexContent>
</xs:complexType>
<!--
##############################################################################################

BEGIN Ancilliary TYPES
##############################################################################################

-->
<!--
#### TYPE #### MetaDataType
-->
<xs:complexType name="MetaDataType">
  <xs:annotation>
    <xs:documentation>Uses meta data specification of the dublin core: http://
        dublincore.org/usage/terms/dc/current-elements/</xs:documentation>
  </xs:annotation>
  <xs:sequence>
    <xs:element name="title" type="xs:normalizedString" minOccurs="0"/>
    <xs:element name="creator" type="xs:string" minOccurs="0" maxOccurs="unbounded
        "/>
```

```
    <xs:element name="subject" type="xs:string" minOccurs="0" maxOccurs="unbounded
        "/>
    <xs:element name="description" type="xs:normalizedString" minOccurs="0"/>
    <xs:element name="contributor" type="xs:string" minOccurs="0" maxOccurs="
        unbounded"/>
    <xs:element name="coverage" type="xs:string" minOccurs="0"/>
    <xs:element name="validFrom" type="xs:date" minOccurs="0"/>
    <xs:element name="validUntil" type="xs:date" minOccurs="0"/>
    <xs:element name="created" type="xs:date" minOccurs="0"/>
    <xs:element name="version" type="xs:decimal" minOccurs="0"/>
    <xs:element name="status" type="xs:string" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>
<!--
#### TYPE #### IndividualOrOrganisationType
-->
<xs:complexType name="IndividualOrOrganisationType">
  <xs:sequence>
    <xs:element name="name" type="xs:string"/>
    <xs:choice maxOccurs="4">
      <xs:element name="address" type="yawl:AddressType"/>
      <xs:element name="emailAddress" type="xs:anyURI"/>
    </xs:choice>
    <xs:element name="description" type="xs:normalizedString" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>
<!--
#### TYPE #### AddressType
-->
<xs:complexType name="AddressType">
  <xs:sequence>
    <xs:choice>
      <xs:element name="numAndStreet">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="number">
              <xs:simpleType>
                <xs:restriction base="xs:string">
                  <xs:pattern value="\d*(/|-)*\d+"/>
                </xs:restriction>
              </xs:simpleType>
            </xs:element>
            <xs:element name="street">
              <xs:simpleType>
                <xs:restriction base="xs:string">
                  <xs:pattern value="\D+\s+[S|s|R|r|A|a|C|c|B|b|o](treet|oad|venue
                      |t|d|ve|rescent|lvd|ther)"/>
                </xs:restriction>
              </xs:simpleType>
            </xs:element>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
```

```
        <xs:element name="GPOBox" type="xs:positiveInteger"/>
      </xs:choice>
      <xs:element name="suburbOrTown" type="xs:string" minOccurs="0"/>
      <xs:element name="postOrZipCode">
        <xs:simpleType>
          <xs:restriction base="xs:positiveInteger">
            <xs:minInclusive value="1000"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      <xs:element name="state" type="xs:string" minOccurs="0"/>
      <xs:element name="country" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
  <!-- #### TYPE ####  ImportType for importing Schmeas from other documents -->
  <xs:complexType name="ImportType">
    <xs:attribute name="namespace" type="xs:anyURI"/>
    <xs:attribute name="location" type="xs:anyURI"/>
  </xs:complexType>
  <!-- #### TYPE #### SchemaType for declaring data types to use in YAWL - using XML
       Schema -->
  <xs:complexType name="SchemaType">
    <xs:sequence>
      <xs:element name="type" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:any namespace="http://www.w3.org/2001/XMLSchema" processContents="
                lax" maxOccurs="unbounded"/>
          </xs:sequence>
          <xs:attribute name="targetNamespace" type="xs:anyURI"/>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

Listing A.2 – Schéma XML du fichier de modélisation (après modifications)

Annexe B

Fichier XML de modélisation

B.1 Exemple 1

Cet exemple montre le processus de réservation d'un voyage par une agence de voyages. Après l'enregistrement de la demande d'un voyage, il faut réserver un vol, un hôtel, et/ou une voiture. Enfin, on effectue le payement. La figure B.1 montre ce workflow en YAWL.

B.1.1 Modélisation

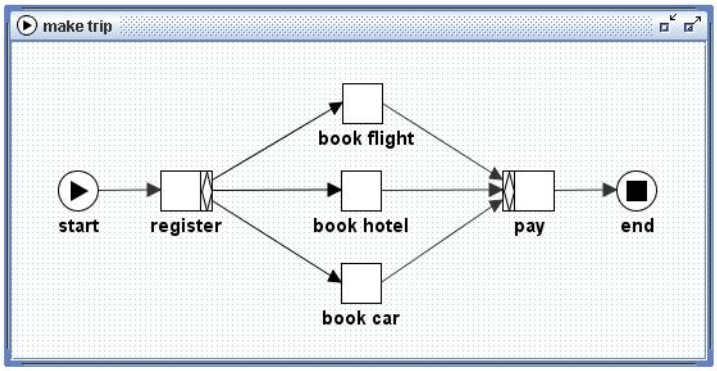

Figure B.1 – Modélisation de l'exemple 1

B.1.2 Fichier XML

Le listing B.1 montre le workflow précédent en XML.

```
<?xml version="1.0" encoding="UTF-8"?>
<specificationSet xmlns="http://www.citi.qut.edu.au/yawl" xmlns:xsi="http://www.w3.
    org/2001/XMLSchema-instance" version="Beta 7.1" xsi:schemaLocation="http://www.
    citi.qut.edu.au/yawl d:/yawl/schema/YAWL_SchemaBeta7.1.xsd">
  <specification uri="makeTrip1_1.3.ywl">
    <metaData>
      <title>make trip 1 process</title>
      <creator>C Ouyang</creator>
```

```
  <description>A simple process that provides a trip booking service with a
      single leg</description>
  <version>0.1</version>
</metaData>
<schema xmlns="http://www.w3.org/2001/XMLSchema">
  <complexType name="tripRegistrationType">
    <sequence>
      <element maxOccurs="1" minOccurs="1" name="startDate">
        <complexType>
          <sequence maxOccurs="1" minOccurs="1">
            <element name="year" type="long" />
            <element name="month" type="long" />
            <element name="day" type="long" />
          </sequence>
        </complexType>
      </element>
      <element maxOccurs="1" minOccurs="1" name="endDate">
        <complexType>
          <sequence maxOccurs="1" minOccurs="1">
            <element name="year" type="long" />
            <element name="month" type="long" />
            <element name="day" type="long" />
          </sequence>
        </complexType>
      </element>
      <element name="want_flight" type="boolean" />
      <element name="want_hotel" type="boolean" />
      <element name="want_car" type="boolean" />
      <element name="payAccNumber" type="string" />
    </sequence>
  </complexType>
  <complexType name="dateType">
    <sequence maxOccurs="1" minOccurs="1">
      <element name="year" type="long" />
      <element name="month" type="long" />
      <element name="day" type="long" />
    </sequence>
  </complexType>
</schema>
<decomposition id="make_trip" isRootNet="true" xsi:type="NetFactsType">
  <localVariable>
    <name>carDetails</name>
    <type>string</type>
    <namespace>http://www.w3.org/2001/XMLSchema</namespace>
    <initialValue>n/a</initialValue>
  </localVariable>
  <localVariable>
    <name>customer</name>
    <type>string</type>
    <namespace>http://www.w3.org/2001/XMLSchema</namespace>
    <initialValue>Type name of customer</initialValue>
  </localVariable>
  <localVariable>
```

```
  <name>flightDetails</name>
  <type>string</type>
  <namespace>http://www.w3.org/2001/XMLSchema</namespace>
  <initialValue>n/a</initialValue>
</localVariable>
<localVariable>
  <name>hotelDetails</name>
  <type>string</type>
  <namespace>http://www.w3.org/2001/XMLSchema</namespace>
  <initialValue>n/a</initialValue>
</localVariable>
<localVariable>
  <name>registrInfo</name>
  <type>tripRegistrationType</type>
  <namespace>http://www.w3.org/2001/XMLSchema</namespace>
  <initialValue />
</localVariable>
<processControlElements>
  <inputCondition id="InputCondition_1">
    <name>start</name>
    <flowsInto>
      <nextElementRef id="register_7" />
    </flowsInto>
  </inputCondition>
  <task id="register_7">
    <name>register</name>
    <flowsInto>
      <nextElementRef id="book_hotel_5" />
      <predicate>/make_trip/registrInfo/want_hotel='true'</predicate>
    </flowsInto>
    <flowsInto>
      <nextElementRef id="book_car_3" />
      <predicate>/make_trip/registrInfo/want_car='true'</predicate>
      <isDefaultFlow />
    </flowsInto>
    <flowsInto>
      <nextElementRef id="book_flight_4" />
      <predicate>/make_trip/registrInfo/want_flight='true'</predicate>
    </flowsInto>
    <join code="xor" />
    <split code="or" />
    <startingMappings>
      <mapping>
        <expression query="&lt;customer&gt;{/make_trip/customer/text()}&lt;/
            customer&gt;" />
        <mapsTo>customer</mapsTo>
      </mapping>
    </startingMappings>
    <completedMappings>
      <mapping>
        <expression query="&lt;registrInfo&gt;{/register/registrInfo/*}&lt;/
            registrInfo&gt;" />
        <mapsTo>registrInfo</mapsTo>
```

```
      </mapping>
      <mapping>
        <expression query="&lt;customer&gt;{/register/customer/text()}&lt;/
            customer&gt;" />
        <mapsTo>customer</mapsTo>
      </mapping>
    </completedMappings>
    <decomposesTo id="register" />
  </task>
  <task id="book_hotel_5">
    <name>book hotel</name>
    <flowsInto>
      <nextElementRef id="pay_6" />
    </flowsInto>
    <join code="xor" />
    <split code="and" />
    <startingMappings>
      <mapping>
        <expression query="&lt;startDate&gt;{/make_trip/registrInfo/startDate
            /*}&lt;/startDate&gt;" />
        <mapsTo>startDate</mapsTo>
      </mapping>
      <mapping>
        <expression query="&lt;customer&gt;{/make_trip/customer/text()}&lt;/
            customer&gt;" />
        <mapsTo>customer</mapsTo>
      </mapping>
      <mapping>
        <expression query="&lt;endDate&gt;{/make_trip/registrInfo/endDate/*}&
            lt;/endDate&gt;" />
        <mapsTo>endDate</mapsTo>
      </mapping>
    </startingMappings>
    <completedMappings>
      <mapping>
        <expression query="&lt;hotelDetails&gt;{/book_hotel/hotelDetails/text
            ()}&lt;/hotelDetails&gt;" />
        <mapsTo>hotelDetails</mapsTo>
      </mapping>
    </completedMappings>
    <decomposesTo id="book_hotel" />
  </task>
  <task id="book_flight_4">
    <name>book flight</name>
    <flowsInto>
      <nextElementRef id="pay_6" />
    </flowsInto>
    <join code="xor" />
    <split code="and" />
    <startingMappings>
      <mapping>
        <expression query="&lt;startDate&gt;{/make_trip/registrInfo/startDate
            /*}&lt;/startDate&gt;" />
```

```
        <mapsTo>startDate</mapsTo>
      </mapping>
      <mapping>
        <expression query="&lt;customer&gt;{/make_trip/customer/text()}&lt;/
            customer&gt;" />
        <mapsTo>customer</mapsTo>
      </mapping>
      <mapping>
        <expression query="&lt;endDate&gt;{/make_trip/registrInfo/endDate/*}&
            lt;/endDate&gt;" />
        <mapsTo>endDate</mapsTo>
      </mapping>
    </startingMappings>
    <completedMappings>
      <mapping>
        <expression query="&lt;flightDetails&gt;{/book_flight/flightDetails/
            text()}&lt;/flightDetails&gt;" />
        <mapsTo>flightDetails</mapsTo>
      </mapping>
    </completedMappings>
    <decomposesTo id="book_flight" />
  </task>
  <task id="book_car_3">
    <name>book car</name>
    <flowsInto>
      <nextElementRef id="pay_6" />
    </flowsInto>
    <join code="xor" />
    <split code="and" />
    <startingMappings>
      <mapping>
        <expression query="&lt;startDate&gt;{/make_trip/registrInfo/startDate
            /*}&lt;/startDate&gt;" />
        <mapsTo>startDate</mapsTo>
      </mapping>
      <mapping>
        <expression query="&lt;customer&gt;{/make_trip/customer/text()}&lt;/
            customer&gt;" />
        <mapsTo>customer</mapsTo>
      </mapping>
      <mapping>
        <expression query="&lt;endDate&gt;{/make_trip/registrInfo/endDate/*}&
            lt;/endDate&gt;" />
        <mapsTo>endDate</mapsTo>
      </mapping>
    </startingMappings>
    <completedMappings>
      <mapping>
        <expression query="&lt;carDetails&gt;{/book_car/carDetails/text()}&lt
            ;/carDetails&gt;" />
        <mapsTo>carDetails</mapsTo>
      </mapping>
    </completedMappings>
```

```
      <decomposesTo id="book_car" />
    </task>
    <task id="pay_6">
      <name>pay</name>
      <flowsInto>
        <nextElementRef id="OutputCondition_2" />
      </flowsInto>
      <join code="or" />
      <split code="and" />
      <startingMappings>
        <mapping>
          <expression query="&lt;flightDetails&gt;{/make_trip/flightDetails/text
              ()}&lt;/flightDetails&gt;" />
          <mapsTo>flightDetails</mapsTo>
        </mapping>
        <mapping>
          <expression query="&lt;customer&gt;{/make_trip/customer/text()}&lt;/
              customer&gt;" />
          <mapsTo>customer</mapsTo>
        </mapping>
        <mapping>
          <expression query="&lt;payAccNumber&gt;{/make_trip/registrInfo/
              payAccNumber/text()}&lt;/payAccNumber&gt;" />
          <mapsTo>payAccNumber</mapsTo>
        </mapping>
        <mapping>
          <expression query="&lt;hotelDetails&gt;{/make_trip/hotelDetails/text()
              }&lt;/hotelDetails&gt;" />
          <mapsTo>hotelDetails</mapsTo>
        </mapping>
        <mapping>
          <expression query="&lt;carDetails&gt;{/make_trip/carDetails/text()}&lt
              ;/carDetails&gt;" />
          <mapsTo>carDetails</mapsTo>
        </mapping>
      </startingMappings>
      <decomposesTo id="pay" />
    </task>
    <outputCondition id="OutputCondition_2">
      <name>end</name>
    </outputCondition>
  </processControlElements>
</decomposition>
<decomposition id="pay" xsi:type="WebServiceGatewayFactsType">
  <inputParam>
    <name>customer</name>
    <type>string</type>
    <namespace>http://www.w3.org/2001/XMLSchema</namespace>
  </inputParam>
  <inputParam>
    <name>flightDetails</name>
    <type>string</type>
    <namespace>http://www.w3.org/2001/XMLSchema</namespace>
```

```
    </inputParam>
    <inputParam>
      <name>hotelDetails</name>
      <type>string</type>
      <namespace>http://www.w3.org/2001/XMLSchema</namespace>
    </inputParam>
    <inputParam>
      <name>carDetails</name>
      <type>string</type>
      <namespace>http://www.w3.org/2001/XMLSchema</namespace>
    </inputParam>
    <inputParam>
      <name>payAccNumber</name>
      <type>string</type>
      <namespace>http://www.w3.org/2001/XMLSchema</namespace>
    </inputParam>
  </decomposition>
  <decomposition id="book_car" xsi:type="WebServiceGatewayFactsType">
    <inputParam>
      <name>customer</name>
      <type>string</type>
      <namespace>http://www.w3.org/2001/XMLSchema</namespace>
    </inputParam>
    <inputParam>
      <name>startDate</name>
      <type>dateType</type>
      <namespace>http://www.w3.org/2001/XMLSchema</namespace>
    </inputParam>
    <inputParam>
      <name>endDate</name>
      <type>dateType</type>
      <namespace>http://www.w3.org/2001/XMLSchema</namespace>
    </inputParam>
    <outputParam>
      <name>carDetails</name>
      <type>string</type>
      <namespace>http://www.w3.org/2001/XMLSchema</namespace>
    </outputParam>
  </decomposition>
  <decomposition id="book_flight" xsi:type="WebServiceGatewayFactsType">
    <inputParam>
      <name>customer</name>
      <type>string</type>
      <namespace>http://www.w3.org/2001/XMLSchema</namespace>
    </inputParam>
    <inputParam>
      <name>startDate</name>
      <type>dateType</type>
      <namespace>http://www.w3.org/2001/XMLSchema</namespace>
    </inputParam>
    <inputParam>
      <name>endDate</name>
      <type>dateType</type>
```

```
        <namespace>http://www.w3.org/2001/XMLSchema</namespace>
      </inputParam>
      <outputParam>
        <name>flightDetails</name>
        <type>string</type>
        <namespace>http://www.w3.org/2001/XMLSchema</namespace>
      </outputParam>
    </decomposition>
    <decomposition id="book_hotel" xsi:type="WebServiceGatewayFactsType">
      <inputParam>
        <name>customer</name>
        <type>string</type>
        <namespace>http://www.w3.org/2001/XMLSchema</namespace>
      </inputParam>
      <inputParam>
        <name>startDate</name>
        <type>dateType</type>
        <namespace>http://www.w3.org/2001/XMLSchema</namespace>
      </inputParam>
      <inputParam>
        <name>endDate</name>
        <type>dateType</type>
        <namespace>http://www.w3.org/2001/XMLSchema</namespace>
      </inputParam>
      <outputParam>
        <name>hotelDetails</name>
        <type>string</type>
        <namespace>http://www.w3.org/2001/XMLSchema</namespace>
      </outputParam>
    </decomposition>
    <decomposition id="register" xsi:type="WebServiceGatewayFactsType">
      <inputParam>
        <name>customer</name>
        <type>string</type>
        <namespace>http://www.w3.org/2001/XMLSchema</namespace>
      </inputParam>
      <outputParam>
        <name>registrInfo</name>
        <type>tripRegistrationType</type>
        <namespace>http://www.w3.org/2001/XMLSchema</namespace>
      </outputParam>
      <outputParam>
        <name>customer</name>
        <type>string</type>
        <namespace>http://www.w3.org/2001/XMLSchema</namespace>
      </outputParam>
    </decomposition>
  </specification>
</specificationSet>
```

Listing B.1 – Fichier XML de l'exemple 1

B.2 Exemple 2

L'exemple 2 montre un processus de réservation d'un voyage par une agence de voyages (plus complexe). Après l'enregistrement de la demande d'un voyage, il faut réserver un vol, un hôtel, et/ou une voiture. Etant donné qu'un voyage peut se composer de plusieurs étapes, un sous processus est créé. Il est instancié pour chaque étape du voyage. Enfin, on effectue le payement. La figure B.2 montre ce workflow en YAWL.

B.2.1 Modélisation

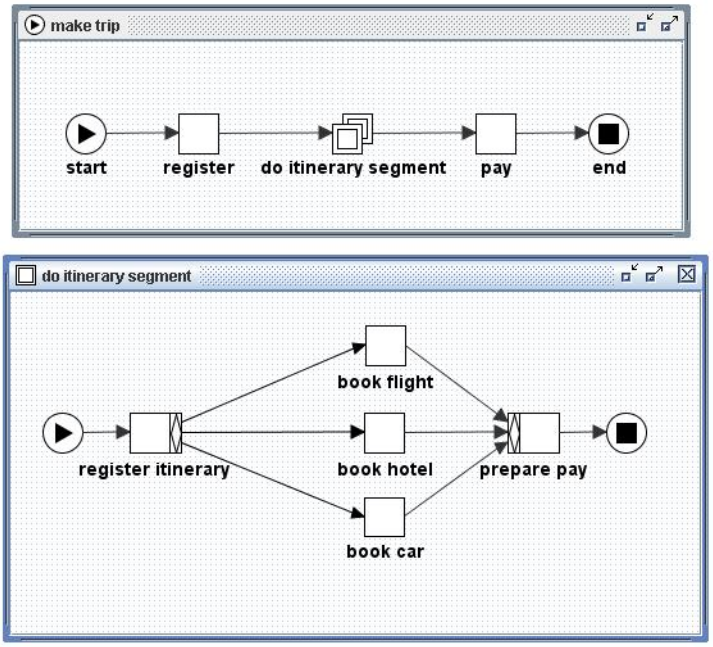

Figure B.2 – Modélisation de l'exemple 2

B.2.2 Fichier XML

Le listing B.2 montre le workflow précédent en XML.

```
<?xml version="1.0" encoding="UTF-8"?>
<specificationSet xmlns="http://www.citi.qut.edu.au/yawl" xmlns:xsi="http://www.w3.
    org/2001/XMLSchema-instance" version="Beta 7.1" xsi:schemaLocation="http://www.
    citi.qut.edu.au/yawl d:/yawl/schema/YAWL_SchemaBeta7.1.xsd">
  <specification uri="makeTrip2_1.3.ywl">
    <metaData>
      <title>make trip 2 process</title>
      <creator>C Ouyang</creator>
      <description>A more complicate process that provides a trip booking service
          with multiple legs</description>
      <version>0.2</version>
    </metaData>
    <schema xmlns="http://www.w3.org/2001/XMLSchema">
      <complexType name="itineraryType">
        <sequence>
```

```
      <element maxOccurs="unbounded" minOccurs="1" name="itinerarySegment">
        <complexType>
          <sequence maxOccurs="1" minOccurs="1">
            <element name="departure_location" type="string" />
            <element name="destination" type="string" />
            <element name="startDate" type="date" />
            <element name="endDate" type="date" />
            <element name="flightDetails" type="string" />
            <element name="hotelDetails" type="string" />
            <element name="carDetails" type="string" />
            <element name="subTotal" type="double" />
          </sequence>
        </complexType>
      </element>
    </sequence>
  </complexType>
  <complexType name="legsType">
    <sequence>
      <element maxOccurs="unbounded" minOccurs="1" name="leg">
        <complexType>
          <sequence maxOccurs="1" minOccurs="1">
            <element name="departure_location" type="string" />
            <element name="destination" type="string" />
          </sequence>
        </complexType>
      </element>
    </sequence>
  </complexType>
  <complexType name="legType">
    <sequence maxOccurs="1" minOccurs="1">
      <element name="departure_location" type="string" />
      <element name="destination" type="string" />
    </sequence>
  </complexType>
  <complexType name="serviceType">
    <sequence maxOccurs="1" minOccurs="1">
      <element name="want_flight" type="boolean" />
      <element name="want_hotel" type="boolean" />
      <element name="want_car" type="boolean" />
    </sequence>
  </complexType>
</schema>
<decomposition id="make_trip" isRootNet="true" xsi:type="NetFactsType">
  <localVariable>
    <name>carDetails</name>
    <type>string</type>
    <namespace>http://www.w3.org/2001/XMLSchema</namespace>
    <initialValue>n/a</initialValue>
  </localVariable>
  <localVariable>
    <name>customer</name>
    <type>string</type>
    <namespace>http://www.w3.org/2001/XMLSchema</namespace>
```

```
  <initialValue>Type name of customer</initialValue>
</localVariable>
<localVariable>
  <name>endDate</name>
  <type>date</type>
  <namespace>http://www.w3.org/2001/XMLSchema</namespace>
  <initialValue>2000-01-01</initialValue>
</localVariable>
<localVariable>
  <name>flightDetails</name>
  <type>string</type>
  <namespace>http://www.w3.org/2001/XMLSchema</namespace>
  <initialValue>n/a</initialValue>
</localVariable>
<localVariable>
  <name>hotelDetails</name>
  <type>string</type>
  <namespace>http://www.w3.org/2001/XMLSchema</namespace>
  <initialValue>n/a</initialValue>
</localVariable>
<localVariable>
  <name>itinerary</name>
  <type>itineraryType</type>
  <namespace>http://www.w3.org/2001/XMLSchema</namespace>
  <initialValue />
</localVariable>
<localVariable>
  <name>legs</name>
  <type>legsType</type>
  <namespace>http://www.w3.org/2001/XMLSchema</namespace>
  <initialValue />
</localVariable>
<localVariable>
  <name>payAccNumber</name>
  <type>string</type>
  <namespace>http://www.w3.org/2001/XMLSchema</namespace>
  <initialValue />
</localVariable>
<localVariable>
  <name>startDate</name>
  <type>date</type>
  <namespace>http://www.w3.org/2001/XMLSchema</namespace>
  <initialValue>2000-01-01</initialValue>
</localVariable>
<localVariable>
  <name>subTotal</name>
  <type>double</type>
  <namespace>http://www.w3.org/2001/XMLSchema</namespace>
  <initialValue>0.00</initialValue>
</localVariable>
<processControlElements>
  <inputCondition id="InputCondition_14">
    <name>start</name>
```

```
  <flowsInto>
    <nextElementRef id="register_16" />
  </flowsInto>
</inputCondition>
<task id="register_16">
  <name>register</name>
  <flowsInto>
    <nextElementRef id="do_itinerary_segment_18" />
  </flowsInto>
  <join code="xor" />
  <split code="and" />
  <startingMappings>
    <mapping>
      <expression query="&lt;customer&gt;{/make_trip/customer/text()}&lt;/
          customer&gt;" />
      <mapsTo>customer</mapsTo>
    </mapping>
  </startingMappings>
  <completedMappings>
    <mapping>
      <expression query="&lt;payAccNumber&gt;{/register/payAccNumber/text()
          }&lt;/payAccNumber&gt;" />
      <mapsTo>payAccNumber</mapsTo>
    </mapping>
    <mapping>
      <expression query="&lt;customer&gt;{/register/customer/text()}&lt;/
          customer&gt;" />
      <mapsTo>customer</mapsTo>
    </mapping>
    <mapping>
      <expression query="&lt;legs&gt;{/register/legs/*}&lt;/legs&gt;" />
      <mapsTo>legs</mapsTo>
    </mapping>
  </completedMappings>
  <decomposesTo id="register" />
</task>
<task id="do_itinerary_segment_18" xsi:type="
    MultipleInstanceExternalTaskFactsType">
  <name>do itinerary segment</name>
  <flowsInto>
    <nextElementRef id="pay_17" />
  </flowsInto>
  <join code="xor" />
  <split code="and" />
  <startingMappings>
    <mapping>
      <expression query="&lt;flightDetails&gt;{/make_trip/flightDetails/text
          ()}&lt;/flightDetails&gt;" />
      <mapsTo>flightDetails</mapsTo>
    </mapping>
    <mapping>
      <expression query="&lt;startDate&gt;{/make_trip/startDate/text()}&lt;/
          startDate&gt;" />
```

```
      <mapsTo>startDate</mapsTo>
    </mapping>
    <mapping>
      <expression query="&lt;subTotal&gt;{number(/make_trip/subTotal/text())
          }&lt;/subTotal&gt;" />
      <mapsTo>subTotal</mapsTo>
    </mapping>
    <mapping>
      <expression query="&lt;customer&gt;{/make_trip/customer/text()}&lt;/
          customer&gt;" />
      <mapsTo>customer</mapsTo>
    </mapping>
    <mapping>
      <expression query="&lt;endDate&gt;{/make_trip/endDate/text()}&lt;/
          endDate&gt;" />
      <mapsTo>endDate</mapsTo>
    </mapping>
    <mapping>
      <expression query="&lt;hotelDetails&gt;{/make_trip/hotelDetails/text()
          }&lt;/hotelDetails&gt;" />
      <mapsTo>hotelDetails</mapsTo>
    </mapping>
    <mapping>
      <expression query="&lt;carDetails&gt;{/make_trip/carDetails/text()}&lt
          ;/carDetails&gt;" />
      <mapsTo>carDetails</mapsTo>
    </mapping>
  </startingMappings>
  <decomposesTo id="do_itinerary_segment" />
  <minimum>1</minimum>
  <maximum>10</maximum>
  <threshold>10</threshold>
  <creationMode code="static" />
  <miDataInput>
    <expression query="/make_trip/legs" />
    <splittingExpression query="for $d in /legs/* return $d" />
    <formalInputParam>leg</formalInputParam>
  </miDataInput>
  <miDataOutput>
    <formalOutputExpression query="&lt;itinerarySegment&gt; { /
        do_itinerary_segment/leg/departure_location } { /
        do_itinerary_segment/leg/destination } { /do_itinerary_segment/
        startDate } { /do_itinerary_segment/endDate } { if(/
        do_itinerary_segment/flightDetails/text())   then /
        do_itinerary_segment/flightDetails else ()} { if(/
        do_itinerary_segment/hotelDetails/text())   then /
        do_itinerary_segment/hotelDetails else() } { if(/
        do_itinerary_segment/carDetails/text())   then /do_itinerary_segment
        /carDetails else() } { /do_itinerary_segment/subTotal } &lt;/
        itinerarySegment&gt;" />
    <outputJoiningExpression query="&lt;itinerary&gt;{for $d in /
        do_itinerary_segment/itinerarySegment return $d}&lt;/itinerary&gt;"
        />
```

```
          <resultAppliedToLocalVariable>itinerary</resultAppliedToLocalVariable>
        </miDataOutput>
      </task>
      <task id="pay_17">
        <name>pay</name>
        <flowsInto>
          <nextElementRef id="OutputCondition_15" />
        </flowsInto>
        <join code="xor" />
        <split code="and" />
        <startingMappings>
          <mapping>
            <expression query="&lt;total&gt;{sum(/make_trip/itinerary/
                itinerarySegment/subTotal)} &lt;/total&gt;" />
            <mapsTo>total</mapsTo>
          </mapping>
          <mapping>
            <expression query="&lt;customer&gt;{/make_trip/customer/text()}&lt;/
                customer&gt;" />
            <mapsTo>customer</mapsTo>
          </mapping>
          <mapping>
            <expression query="&lt;payAccNumber&gt;{/make_trip/payAccNumber/text()
                }&lt;/payAccNumber&gt;" />
            <mapsTo>payAccNumber</mapsTo>
          </mapping>
          <mapping>
            <expression query="&lt;itinerary&gt;{/make_trip/itinerary/*}&lt;/
                itinerary&gt;" />
            <mapsTo>itinerary</mapsTo>
          </mapping>
        </startingMappings>
        <decomposesTo id="pay" />
      </task>
      <outputCondition id="OutputCondition_15">
        <name>end</name>
      </outputCondition>
    </processControlElements>
  </decomposition>
  <decomposition id="do_itinerary_segment" xsi:type="NetFactsType">
    <inputParam>
      <name>customer</name>
      <type>string</type>
      <namespace>http://www.w3.org/2001/XMLSchema</namespace>
    </inputParam>
    <inputParam>
      <name>leg</name>
      <type>legType</type>
      <namespace>http://www.w3.org/2001/XMLSchema</namespace>
    </inputParam>
    <inputParam>
      <name>startDate</name>
      <type>date</type>
```

```
  <namespace>http://www.w3.org/2001/XMLSchema</namespace>
</inputParam>
<inputParam>
  <name>endDate</name>
  <type>date</type>
  <namespace>http://www.w3.org/2001/XMLSchema</namespace>
</inputParam>
<inputParam>
  <name>flightDetails</name>
  <type>string</type>
  <namespace>http://www.w3.org/2001/XMLSchema</namespace>
</inputParam>
<inputParam>
  <name>hotelDetails</name>
  <type>string</type>
  <namespace>http://www.w3.org/2001/XMLSchema</namespace>
</inputParam>
<inputParam>
  <name>carDetails</name>
  <type>string</type>
  <namespace>http://www.w3.org/2001/XMLSchema</namespace>
</inputParam>
<inputParam>
  <name>subTotal</name>
  <type>double</type>
  <namespace>http://www.w3.org/2001/XMLSchema</namespace>
</inputParam>
<outputParam>
  <name>flightDetails</name>
  <type>string</type>
  <namespace>http://www.w3.org/2001/XMLSchema</namespace>
</outputParam>
<outputParam>
  <name>startDate</name>
  <type>date</type>
  <namespace>http://www.w3.org/2001/XMLSchema</namespace>
</outputParam>
<outputParam>
  <name>leg</name>
  <type>legType</type>
  <namespace>http://www.w3.org/2001/XMLSchema</namespace>
</outputParam>
<outputParam>
  <name>subTotal</name>
  <type>double</type>
  <namespace>http://www.w3.org/2001/XMLSchema</namespace>
</outputParam>
<outputParam>
  <name>customer</name>
  <type>string</type>
  <namespace>http://www.w3.org/2001/XMLSchema</namespace>
</outputParam>
<outputParam>
```

```
  <name>endDate</name>
  <type>date</type>
  <namespace>http://www.w3.org/2001/XMLSchema</namespace>
</outputParam>
<outputParam>
  <name>hotelDetails</name>
  <type>string</type>
  <namespace>http://www.w3.org/2001/XMLSchema</namespace>
</outputParam>
<outputParam>
  <name>carDetails</name>
  <type>string</type>
  <namespace>http://www.w3.org/2001/XMLSchema</namespace>
</outputParam>
<localVariable>
  <name>serviceRequired</name>
  <type>serviceType</type>
  <namespace>http://www.w3.org/2001/XMLSchema</namespace>
  <initialValue />
</localVariable>
<processControlElements>
  <inputCondition id="InputCondition_1">
    <flowsInto>
      <nextElementRef id="register_itinerary_3" />
    </flowsInto>
  </inputCondition>
  <task id="register_itinerary_3">
    <name>register itinerary</name>
    <flowsInto>
      <nextElementRef id="book_flight_6" />
      <predicate>/do_itinerary_segment/serviceRequired/want_flight='true'</
          predicate>
    </flowsInto>
    <flowsInto>
      <nextElementRef id="book_hotel_4" />
      <predicate>/do_itinerary_segment/serviceRequired/want_hotel='true'</
          predicate>
    </flowsInto>
    <flowsInto>
      <nextElementRef id="book_car_7" />
      <predicate>/do_itinerary_segment/serviceRequired/want_car='true'</
          predicate>
      <isDefaultFlow />
    </flowsInto>
    <join code="xor" />
    <split code="or" />
    <startingMappings>
      <mapping>
        <expression query="&lt;leg&gt;{/do_itinerary_segment/leg/*}&lt;/leg&gt
            ;" />
        <mapsTo>leg</mapsTo>
      </mapping>
      <mapping>
```

```
        <expression query="&lt;customer&gt;{/do_itinerary_segment/customer/
            text()}&lt;/customer&gt;" />
        <mapsTo>customer</mapsTo>
      </mapping>
    </startingMappings>
    <completedMappings>
      <mapping>
        <expression query="&lt;endDate&gt;{/register_itinerary/endDate/text()
            }&lt;/endDate&gt;" />
        <mapsTo>endDate</mapsTo>
      </mapping>
      <mapping>
        <expression query="&lt;serviceRequired&gt;{/register_itinerary/
            serviceRequired/*}&lt;/serviceRequired&gt;" />
        <mapsTo>serviceRequired</mapsTo>
      </mapping>
      <mapping>
        <expression query="&lt;startDate&gt;{/register_itinerary/startDate/
            text()}&lt;/startDate&gt;" />
        <mapsTo>startDate</mapsTo>
      </mapping>
    </completedMappings>
    <decomposesTo id="register_itinerary" />
  </task>
  <task id="book_car_7">
    <name>book car</name>
    <flowsInto>
      <nextElementRef id="prepare_pay_5" />
    </flowsInto>
    <join code="xor" />
    <split code="and" />
    <startingMappings>
      <mapping>
        <expression query="&lt;startDate&gt;{/do_itinerary_segment/startDate/
            text()}&lt;/startDate&gt;" />
        <mapsTo>startDate</mapsTo>
      </mapping>
      <mapping>
        <expression query="&lt;leg&gt;{/do_itinerary_segment/leg/*}&lt;/leg&gt
            ;" />
        <mapsTo>leg</mapsTo>
      </mapping>
      <mapping>
        <expression query="&lt;customer&gt;{/do_itinerary_segment/customer/
            text()}&lt;/customer&gt;" />
        <mapsTo>customer</mapsTo>
      </mapping>
      <mapping>
        <expression query="&lt;endDate&gt;{/do_itinerary_segment/endDate/text
            ()}&lt;/endDate&gt;" />
        <mapsTo>endDate</mapsTo>
      </mapping>
    </startingMappings>
```

```
  <completedMappings>
    <mapping>
      <expression query="&lt;carDetails&gt;{/book_car/carDetails/text()}&lt
          ;/carDetails&gt;" />
      <mapsTo>carDetails</mapsTo>
    </mapping>
  </completedMappings>
  <decomposesTo id="book_car" />
</task>
<task id="book_flight_6">
  <name>book flight</name>
  <flowsInto>
    <nextElementRef id="prepare_pay_5" />
  </flowsInto>
  <join code="xor" />
  <split code="and" />
  <startingMappings>
    <mapping>
      <expression query="&lt;startDate&gt;{/do_itinerary_segment/startDate/
          text()}&lt;/startDate&gt;" />
      <mapsTo>startDate</mapsTo>
    </mapping>
    <mapping>
      <expression query="&lt;leg&gt;{/do_itinerary_segment/leg/*}&lt;/leg&gt
          ;" />
      <mapsTo>leg</mapsTo>
    </mapping>
    <mapping>
      <expression query="&lt;customer&gt;{/do_itinerary_segment/customer/
          text()}&lt;/customer&gt;" />
      <mapsTo>customer</mapsTo>
    </mapping>
    <mapping>
      <expression query="&lt;endDate&gt;{/do_itinerary_segment/endDate/text
          ()}&lt;/endDate&gt;" />
      <mapsTo>endDate</mapsTo>
    </mapping>
  </startingMappings>
  <completedMappings>
    <mapping>
      <expression query="&lt;flightDetails&gt;{/book_flight/flightDetails/
          text()}&lt;/flightDetails&gt;" />
      <mapsTo>flightDetails</mapsTo>
    </mapping>
  </completedMappings>
  <decomposesTo id="book_flight" />
</task>
<task id="book_hotel_4">
  <name>book hotel</name>
  <flowsInto>
    <nextElementRef id="prepare_pay_5" />
  </flowsInto>
  <join code="xor" />
```

```
  <split code="and" />
  <startingMappings>
    <mapping>
      <expression query="&lt;startDate&gt;{/do_itinerary_segment/startDate/
          text()}&lt;/startDate&gt;" />
      <mapsTo>startDate</mapsTo>
    </mapping>
    <mapping>
      <expression query="&lt;leg&gt;{/do_itinerary_segment/leg/*}&lt;/leg&gt
          ;" />
      <mapsTo>leg</mapsTo>
    </mapping>
    <mapping>
      <expression query="&lt;customer&gt;{/do_itinerary_segment/customer/
          text()}&lt;/customer&gt;" />
      <mapsTo>customer</mapsTo>
    </mapping>
    <mapping>
      <expression query="&lt;endDate&gt;{/do_itinerary_segment/endDate/text
          ()}&lt;/endDate&gt;" />
      <mapsTo>endDate</mapsTo>
    </mapping>
  </startingMappings>
  <completedMappings>
    <mapping>
      <expression query="&lt;hotelDetails&gt;{/book_hotel/hotelDetails/text
          ()}&lt;/hotelDetails&gt;" />
      <mapsTo>hotelDetails</mapsTo>
    </mapping>
  </completedMappings>
  <decomposesTo id="book_hotel" />
</task>
<task id="prepare_pay_5">
  <name>prepare pay</name>
  <flowsInto>
    <nextElementRef id="OutputCondition_2" />
  </flowsInto>
  <join code="or" />
  <split code="and" />
  <startingMappings>
    <mapping>
      <expression query="&lt;flightDetails&gt;{/do_itinerary_segment/
          flightDetails/text()}&lt;/flightDetails&gt;" />
      <mapsTo>flightDetails</mapsTo>
    </mapping>
    <mapping>
      <expression query="&lt;customer&gt;{/do_itinerary_segment/customer/
          text()}&lt;/customer&gt;" />
      <mapsTo>customer</mapsTo>
    </mapping>
    <mapping>
      <expression query="&lt;hotelDetails&gt;{/do_itinerary_segment/
          hotelDetails/text()}&lt;/hotelDetails&gt;" />
```

```
          <mapsTo>hotelDetails</mapsTo>
        </mapping>
        <mapping>
          <expression query="&lt;carDetails&gt;{/do_itinerary_segment/carDetails
              /text()}&lt;/carDetails&gt;" />
          <mapsTo>carDetails</mapsTo>
        </mapping>
      </startingMappings>
      <completedMappings>
        <mapping>
          <expression query="&lt;subTotal&gt;{number(/prepare_pay/subTotal/text
              ())}&lt;/subTotal&gt;" />
          <mapsTo>subTotal</mapsTo>
        </mapping>
      </completedMappings>
      <decomposesTo id="prepare_pay" />
    </task>
    <outputCondition id="OutputCondition_2" />
  </processControlElements>
</decomposition>
<decomposition id="pay" xsi:type="WebServiceGatewayFactsType">
  <inputParam>
    <name>customer</name>
    <type>string</type>
    <namespace>http://www.w3.org/2001/XMLSchema</namespace>
  </inputParam>
  <inputParam>
    <name>itinerary</name>
    <type>itineraryType</type>
    <namespace>http://www.w3.org/2001/XMLSchema</namespace>
  </inputParam>
  <inputParam>
    <name>total</name>
    <type>double</type>
    <namespace>http://www.w3.org/2001/XMLSchema</namespace>
  </inputParam>
  <inputParam>
    <name>payAccNumber</name>
    <type>string</type>
    <namespace>http://www.w3.org/2001/XMLSchema</namespace>
  </inputParam>
</decomposition>
<decomposition id="book_car" xsi:type="WebServiceGatewayFactsType">
  <inputParam>
    <name>customer</name>
    <type>string</type>
    <namespace>http://www.w3.org/2001/XMLSchema</namespace>
  </inputParam>
  <inputParam>
    <name>leg</name>
    <type>legType</type>
    <namespace>http://www.w3.org/2001/XMLSchema</namespace>
  </inputParam>
```

```
    <inputParam>
      <name>startDate</name>
      <type>date</type>
      <namespace>http://www.w3.org/2001/XMLSchema</namespace>
    </inputParam>
    <inputParam>
      <name>endDate</name>
      <type>date</type>
      <namespace>http://www.w3.org/2001/XMLSchema</namespace>
    </inputParam>
    <outputParam>
      <name>carDetails</name>
      <type>string</type>
      <namespace>http://www.w3.org/2001/XMLSchema</namespace>
    </outputParam>
  </decomposition>
  <decomposition id="register_itinerary" xsi:type="WebServiceGatewayFactsType">
    <inputParam>
      <name>customer</name>
      <type>string</type>
      <namespace>http://www.w3.org/2001/XMLSchema</namespace>
    </inputParam>
    <inputParam>
      <name>leg</name>
      <type>legType</type>
      <namespace>http://www.w3.org/2001/XMLSchema</namespace>
    </inputParam>
    <outputParam>
      <name>startDate</name>
      <type>date</type>
      <namespace>http://www.w3.org/2001/XMLSchema</namespace>
    </outputParam>
    <outputParam>
      <name>endDate</name>
      <type>date</type>
      <namespace>http://www.w3.org/2001/XMLSchema</namespace>
    </outputParam>
    <outputParam>
      <name>serviceRequired</name>
      <type>serviceType</type>
      <namespace>http://www.w3.org/2001/XMLSchema</namespace>
    </outputParam>
  </decomposition>
  <decomposition id="book_flight" xsi:type="WebServiceGatewayFactsType">
    <inputParam>
      <name>customer</name>
      <type>string</type>
      <namespace>http://www.w3.org/2001/XMLSchema</namespace>
    </inputParam>
    <inputParam>
      <name>leg</name>
      <type>legType</type>
      <namespace>http://www.w3.org/2001/XMLSchema</namespace>
```

```
    </inputParam>
    <inputParam>
      <name>startDate</name>
      <type>date</type>
      <namespace>http://www.w3.org/2001/XMLSchema</namespace>
    </inputParam>
    <inputParam>
      <name>endDate</name>
      <type>date</type>
      <namespace>http://www.w3.org/2001/XMLSchema</namespace>
    </inputParam>
    <outputParam>
      <name>flightDetails</name>
      <type>string</type>
      <namespace>http://www.w3.org/2001/XMLSchema</namespace>
    </outputParam>
  </decomposition>
  <decomposition id="prepare_pay" xsi:type="WebServiceGatewayFactsType">
    <inputParam>
      <name>customer</name>
      <type>string</type>
      <namespace>http://www.w3.org/2001/XMLSchema</namespace>
    </inputParam>
    <inputParam>
      <name>flightDetails</name>
      <type>string</type>
      <namespace>http://www.w3.org/2001/XMLSchema</namespace>
    </inputParam>
    <inputParam>
      <name>hotelDetails</name>
      <type>string</type>
      <namespace>http://www.w3.org/2001/XMLSchema</namespace>
    </inputParam>
    <inputParam>
      <name>carDetails</name>
      <type>string</type>
      <namespace>http://www.w3.org/2001/XMLSchema</namespace>
    </inputParam>
    <outputParam>
      <name>subTotal</name>
      <type>double</type>
      <namespace>http://www.w3.org/2001/XMLSchema</namespace>
    </outputParam>
  </decomposition>
  <decomposition id="book_hotel" xsi:type="WebServiceGatewayFactsType">
    <inputParam>
      <name>customer</name>
      <type>string</type>
      <namespace>http://www.w3.org/2001/XMLSchema</namespace>
    </inputParam>
    <inputParam>
      <name>leg</name>
      <type>legType</type>
```

```
        <namespace>http://www.w3.org/2001/XMLSchema</namespace>
      </inputParam>
      <inputParam>
        <name>startDate</name>
        <type>date</type>
        <namespace>http://www.w3.org/2001/XMLSchema</namespace>
      </inputParam>
      <inputParam>
        <name>endDate</name>
        <type>date</type>
        <namespace>http://www.w3.org/2001/XMLSchema</namespace>
      </inputParam>
      <outputParam>
        <name>hotelDetails</name>
        <type>string</type>
        <namespace>http://www.w3.org/2001/XMLSchema</namespace>
      </outputParam>
    </decomposition>
    <decomposition id="register" xsi:type="WebServiceGatewayFactsType">
      <inputParam>
        <name>customer</name>
        <type>string</type>
        <namespace>http://www.w3.org/2001/XMLSchema</namespace>
      </inputParam>
      <outputParam>
        <name>legs</name>
        <type>legsType</type>
        <namespace>http://www.w3.org/2001/XMLSchema</namespace>
      </outputParam>
      <outputParam>
        <name>customer</name>
        <type>string</type>
        <namespace>http://www.w3.org/2001/XMLSchema</namespace>
      </outputParam>
      <outputParam>
        <name>payAccNumber</name>
        <type>string</type>
        <namespace>http://www.w3.org/2001/XMLSchema</namespace>
      </outputParam>
    </decomposition>
  </specification>
</specificationSet>
```

Listing B.2 – Fichier XML de l'exemple 2

Annexe C

Cœur du workflow

C.1 Connector

C.1.1 AND-Split

La classe *AND-Split* est l'implémentation du connecteur *AND-Split*.

```
1  package connector;
2
3  import reo.core.*;
4  import reo.primitive.*;
5
6  public class AndSplit{
7      private InputEnd in;
8      private OutputEnd left;
9      private OutputEnd right;
10
11     public AndSplit(){
12         Replicator r = new Replicator();
13
14         in = r.getIn();
15         left = r.getLeft();
16         right = r.getRight();
17     }
18
19     public OutputEnd getLeft(){return left;}
20     public OutputEnd getRight(){return right;}
21     public InputEnd getIn(){return in;}
22 }
```

Listing C.1 – Connector - AND-Split

C.1.2 AND-Join

La classe *AND-Join* est l'implémentation du connecteur *AND-Join*.

```
1  package connector;
2
3  import reo.core.*;
4  import reo.primitive.*;
5
6  public class AndJoin{
7      private InputEnd left;
8      private InputEnd right;
9      private OutputEnd out;
10
11     public AndJoin(){
12         SyncDrain sd = new SyncDrain();
13         Replicator replicator = new Replicator();
14
15         sd.getRight().JOIN(replicator.getLeft());
16
17         left = sd.getLeft();
18         right = replicator.getIn();
19         out = replicator.getRight();
20     }
21
22     public InputEnd getLeft(){return left;}
23     public InputEnd getRight(){return right;}
24     public OutputEnd getOut(){return out;}
25 }
```

Listing C.2 – Connector - AND-Join

C.1.3 XOR-Split

La classe *XOR-Split* est l'implémentation du connecteur *XOR-Split*.

```
1  package connector;
2
3  import reo.connector.ExclusiveRouter;
4  import reo.core.*;
5
6  public class XorSplit {
7      private InputEnd in;
8      private OutputEnd left;
9      private OutputEnd right;
10
11     public XorSplit() {
12         ExclusiveRouter er = new ExclusiveRouter();
13
14         in = er.getIn();
15         left = er.getLeft();
16         right = er.getRight();
17     }
18
19     public InputEnd getIn() {return in;}
20     public OutputEnd getLeft() {return left;}
21     public OutputEnd getRight() {return right;}
```

```
22 }
```

Listing C.3 – Connector - XOR-Split

C.1.4 XOR-Join

La classe *XOR-Join* est l'implémentation du connecteur *XOR-Join*.

```
1  package connector;
2
3  import reo.core.*;
4  import reo.primitive.*;
5
6  public class XorJoin{
7      private InputEnd left;
8      private InputEnd right;
9      private OutputEnd out;
10
11     public XorJoin(){
12         Merger merger = new Merger();
13
14         left = merger.getLeft();
15         right = merger.getLeft();
16         out = merger.getOut();
17     }
18
19     public InputEnd getLeft(){return left;}
20     public InputEnd getRight(){return right;}
21     public OutputEnd getOut(){return out;}
22 }
```

Listing C.4 – Connector - XOR-Join

C.1.5 OR-Split

La classe *OR-Split* est l'implémentation du connecteur *OR-Split*.

```
1  package connector;
2
3  import reo.core.*;
4  import reo.primitive.*;
5
6  public class OrSplit {
7      private InputEnd in;
8      private OutputEnd left;
9      private OutputEnd right;
10
11     public OrSplit(){
12         Replicator r = new Replicator();
13         LossySync ls1 = new LossySync();
14         LossySync ls2 = new LossySync();
15
```

```
16          r.getLeft().JOIN(ls1.getLeft());
17          r.getRight().JOIN(ls2.getLeft());
18
19
20          in = r.getIn();
21          left = ls1.getRight();
22          right = ls2.getRight();
23      }
24
25      public InputEnd getIn() {return in;}
26      public OutputEnd getLeft() {return left;}
27      public OutputEnd getRight() {return right;}
28  }
```

Listing C.5 – Connector - OR-Split

C.2 Classes utilitaires

Cette section présente le code Java des différentes classes utilitaires de notre moteur de workflow. L'implémentation réalisée est partielle. Ces classes nous ont permis de vérifier notre implémentation.

C.2.1 InputCondition

La classe *InputCondition* est l'implémentation d'une entrée dans une tâche composée.

```
1  package util;
2
3  import util.Message;
4  import reo.component.Component;
5  import reo.core.InputEnd;
6
7  public class InputCondition extends Thread implements Component{
8      private InputEnd ie;
9      private String name = "InputCondition";
10
11     public InputCondition(InputEnd ie){
12         this.ie = ie;
13     }
14
15     synchronized public void run(){
16         ie.connect(this);
17         System.out.println("Envoi du message à partir de " + name);
18         ie.write(new Message(name, name));
19     }
20
21     public InputEnd getInputEnd(){return ie;}
22 }
```

Listing C.6 – Utilitaire - InputCondition

C.2.2 InstanceAbstract

La classe *InstanceAbstract* est l'implémentation d'une instance abstraite.

```
1  package util;
2
3  public abstract class InstanceAbstract extends Thread{
4      protected int id;
5      protected String name;
6      protected TaskInstanceAbstract mInstance;
7      protected boolean isFinished;
8
9      public boolean isFinished(){return isFinished;}
10 }
```

Listing C.7 – Utilitaire - InstanceAbstract

C.2.3 InstanceAtomic

La classe *InstanceAtomic* est l'implémentation d'une instance d'une tâche atomique.

```
1  package util;
2
3  public class InstanceAtomic extends InstanceAbstract{
4
5      public InstanceAtomic(TaskInstanceAbstract p, String nom, int id){
6          mInstance = p;
7          this.name = nom;
8          this.id = id;
9          start();
10     }
11
12     public void run(){
13         try {
14             Thread.sleep(name.hashCode()*100);
15         } catch (InterruptedException e) {e.printStackTrace();}
16         System.out.println("Tâche " + name + "—" + id + " fini");
17         isFinished = true;
18         if (mInstance.syncrhonization)
19             mInstance.writer();
20     }
21 }
```

Listing C.8 – Utilitaire - InstanceAtomic

C.2.4 InstanceComposite

La classe *InstanceComposite* est l'implémentation d'une instance d'une tâche composée.

```
1  package util;
2
3  public class InstanceComposite extends InstanceAbstract{
```

```
4       private TaskComposite taskComposite;
5
6       public InstanceComposite(TaskInstanceAbstract p, TaskComposite taskComposite,
7                                String nom, int id){
8           mInstance = p;
9           this.taskComposite = taskComposite;
10          this.name = nom;
11          this.id = id;
12          start();
13      }
14
15      public void run(){
16          taskComposite.startWorkflow();
17          System.out.println("Tâche " + name + "—" + id + " fini");
18          while (!taskComposite.isEndTask())
19              ;
20          isFinished = true;
21          if (mInstance.syncrhonization)
22              mInstance.writer();
23      }
24  }
```

Listing C.9 – Utilitaire - InstanceComposite

C.2.5 Message

La classe *Message* est l'implémentation d'un message.

```
1   package util;
2
3   public class Message {
4       public static String NOTASK = "No execute";
5       public static String CANCELTASK = "Cancel task";
6       public static String CANCELWORKFLOW = "Cancel workflow";
7
8       private String name;
9       private String content;
10
11      public Message(String nom, String content){
12          this.name = nom;
13          this.content = content;
14      }
15
16      public String getName(){return name;}
17      public String getContent(){return content;}
18  }
```

Listing C.10 – Utilitaire - Message

C.2.6 ObjectWorkflowAbstract

La classe *ObjectWorkflowAbstract* est l'implémentation commune d'une tâche et d'un état.

```
1  package util;
2
3  import java.util.ArrayList;
4  import reo.component.Component;
5  import reo.core.InputEnd;
6  import reo.core.OutputEnd;
7
8  public abstract class ObjectWorkflowAbstract extends Thread implements Component {
9      protected String id;
10     protected String name;
11     protected String type;
12     protected InputEnd inputEnd;
13     protected OutputEnd outputEnd;
14     protected ArrayList<TaskRelation> relations = new ArrayList<TaskRelation>();
15     // Gestion de la condition pour combler le manque du connecteur Filter
16     protected String condition;
17     // Ne sert quand pour l'exemple, normallement ça devrait être un événement java
18     protected boolean isCancel;
19
20     public String getNameObject(){return name;}
21     public InputEnd getInputEnd(){return inputEnd;}
22     public OutputEnd getOutputEnd(){return outputEnd;}
23
24     public void setInputEnd(InputEnd ie){this.inputEnd = ie;}
25     public void setOutputEnd(OutputEnd oe){this.outputEnd = oe;}
26
27     public void addFlows(String id, String predicate) {
28         TaskRelation f = new TaskRelation(id);
29         f.setPredicate(predicate);
30         relations.add(f);
31     }
32
33     public String getPredicate(String id) {
34         String result = null;
35         for (int i = 0; i < relations.size(); i++) {
36             TaskRelation f = (TaskRelation) relations.get(i);
37             if (f.getNewElementRef().equals(id))
38                 result = f.getPredicate();
39         }
40         System.out.println(result);
41         return result;
42     }
43
44     public void sendCancel(){
45         inputEnd.connect(this);
46         System.out.println("Cancel " + name);
47         inputEnd.write(new Message(name, Message.CANCELTASK));
48     }
49
50     public void sendCancelWorkflow(){
51         inputEnd.connect(this);
52         System.out.println("Cancel Workflow" + name);
53         inputEnd.write(new Message(name, Message.CANCELWORKFLOW));
```

```
54      }
55  }
```

Listing C.11 – Utilitaire - ObjectWorkflowAbstract

C.2.7 OutputCondition

La classe *OutputCondition* est l'implémentation d'une sortie dans une tâche composée.

```
1  package util;
2
3  import reo.component.Component;
4  import reo.core.OutputEnd;
5
6  public class OutputCondition extends Thread implements Component {
7      private OutputEnd oe;
8      private String name = "OutputCondition";;
9
10     public OutputCondition(OutputEnd oe) {
11         this.oe = oe;
12     }
13
14     public void run(){
15         if (oe != null){
16             oe.connect(this);
17             oe.take();
18             System.out.println("Fin de la tâche composé " + name);
19         }
20     }
21 }
```

Listing C.12 – Utilitaire - OutputCondition

C.2.8 TaskAtomic

La classe *TaskAtomic* est l'implémentation d'une tâche atomique.

```
1  package util;
2
3  import util.Message;
4  import reo.core.*;
5
6  public class TaskAtomic extends ObjectWorkflowAbstract {
7
8      public TaskAtomic(OutputEnd oe, InputEnd ie, String nom) {
9          this.inputEnd = ie;
10         this.outputEnd = oe;
11         this.name = nom;
12         this.id = nom;
13         condition = null;
14     }
15
```

```
16      public TaskAtomic(OutputEnd oe, InputEnd ie, String nom, String condition) {
17          this.inputEnd = ie;
18          this.outputEnd = oe;
19          this.name = nom;
20          this.id = nom;
21          this.condition = condition;
22      }
23
24      synchronized public void run() {
25          while (true) {
26              outputEnd.connect(this);
27              Message c1 = (Message) outputEnd.take();
28              System.out.println(name + " a reçu un msg de " + c1.getContent());
29              inputEnd.connect(this);
30              if (c1 != null && c1.getContent().equals(Message.NOTASK)){
31                  System.out.println(Message.NOTASK + name);
32                  inputEnd.write(new Message(name, Message.NOTASK));
33              }
34              if (c1 != null && c1.getContent().equals(Message.CANCELWORKFLOW)){
35                  System.out.println("Cancel " + name);
36                  isCancel = true;
37                  inputEnd.write(new Message(name, Message.CANCELWORKFLOW));
38              }
39              if (isCancel)
40                  return;
41              if (TaskRelation.isPredicate(condition) && c1 != null &&
42                  c1.getContent() != null) {
43                  System.out.println("Envoi message de " + name);
44                  inputEnd.write(new Message(name, name));
45              } else {
46                  System.out.println("Envoi message " + Message.NOTASK + " de " + name
                        );
47                  inputEnd.write(new Message(name, Message.NOTASK));
48              }
49          }
50      }
51  }
```

Listing C.13 – Utilitaire - TaskAtomic

C.2.9 TaskComposite

La classe *TaskCompoite* est l'implémentation d'une tâche composée.

```
1  package util;
2
3  import java.util.ArrayList;
4  import reo.core.InputEnd;
5  import reo.core.OutputEnd;
6  import reo.primitive.SyncChannel;
7
8  public class TaskComposite extends ObjectWorkflowAbstract {
9      private ArrayList<ObjectWorkflowAbstract> tasks;
```

```
10      private InputCondition in;
11      private OutputCondition out;
12      private SyncChannel channelIn = new SyncChannel();
13      private SyncChannel channelOut = new SyncChannel();
14      private boolean endTask = false;
15
16      public TaskComposite() {
17          tasks = new ArrayList<ObjectWorkflowAbstract>();
18          name = "root";
19          in = new InputCondition(channelIn.getLeft());
20          out = new OutputCondition(channelOut.getRight());
21      }
22
23      public TaskComposite(OutputEnd oe, InputEnd ie, String nom) {
24          tasks = new ArrayList<ObjectWorkflowAbstract>();
25          this.name = nom;
26          this.outputEnd = oe;
27          this.inputEnd = ie;
28          in = new InputCondition(channelIn.getLeft());
29          out = new OutputCondition(channelOut.getRight());
30      }
31
32      public void addTask(ObjectWorkflowAbstract t) {
33          if (t.getOutputEnd() == null)
34              t.setOutputEnd(channelIn.getRight());
35          if (t.getInputEnd() == null)
36              t.setInputEnd(channelOut.getLeft());
37          tasks.add(t);
38      }
39
40      synchronized public void run() {
41          if (!name.equals("root")){
42              outputEnd.connect(this);
43              Message c1 = (Message) outputEnd.take();
44              System.out.println(name + " a reçu un msg de " + c1.getContent());
45              inputEnd.connect(this);
46              startWorkflow();
47          }
48          // Attendre la fin de la sous-tâche composée
49          while (out.isAlive())
50              ;
51          endTask = true;
52          if (name.equals("root"))
53              System.out.println("Fin du workflow " + name);
54          else inputEnd.write(new Message(name, name));
55      }
56
57      public boolean isEndTask(){return endTask;}
58
59      public void startWorkflow() {
60          in.start();
61          for (int i = 0; i < tasks.size(); i++)
62              tasks.get(i).start();
```

```
63          out.start();
64          if (name.equals("root"))
65              start();
66      }
67  }
```

Listing C.14 – Utilitaire - TaskComposite

C.2.10 TaskInstanceAbstract

La classe *TaskInstanceAbstract* est l'implémentation de la gestion des instances.

```
1  package util;
2
3  import java.io.BufferedReader;
4  import java.io.IOException;
5  import java.io.InputStream;
6  import java.io.InputStreamReader;
7  import java.util.ArrayList;
8
9  public abstract class TaskInstanceAbstract extends ObjectWorkflowAbstract{
10     protected ArrayList<InstanceAbstract> instances = new ArrayList<InstanceAbstract
           >();
11     public boolean isStatic = true;
12     private int numInstanceStart = -1;
13     public boolean syncrhonization = true;
14
15     synchronized public void run() {
16         InputStream in = System.in;
17         BufferedReader stdin = new BufferedReader(new InputStreamReader(in));
18         while(true){
19             outputEnd.connect(this);
20             outputEnd.take();
21
22             if (numInstanceStart < 0){
23                 try {
24                     System.out.println("Le nombre d'instance au départ ?" );
25                     numInstanceStart = Integer.parseInt(stdin.readLine());
26                     setNumInstanceStart(numInstanceStart);
27                 } catch (IOException e) {e.printStackTrace();}
28             }
29             if (!isStatic){
30                 String rep = "";
31                 while (!rep.equals("n")){
32                     System.out.println("Une nouvelle instance ?" );
33                     try {
34                         rep = stdin.readLine();
35                     } catch (IOException e) {e.printStackTrace();}
36                     if (rep.equals("y"))
37                         newInstance();
38                 }
39                 if (!syncrhonization) writer();
40             }
```

```
41          }
42      }
43
44      public void setNumInstanceStart(int numInstanceStart){
45          this.numInstanceStart = numInstanceStart;
46          for (int i = 0; i < numInstanceStart; i++)
47              newInstance();
48      }
49
50      public void addDynamique(){this.isStatic = false;}
51      public void addStatic (){this.isStatic = true; }
52      public void setSynchronization(boolean b){this.syncrhonization = b;}
53
54      public void writer(){
55          if(syncrhonization){
56              for (int i = 0; i < instances.size(); i++){
57                  System.out.println(instances.get(i).isFinished());
58                  if (!instances.get(i).isFinished())
59                      return;
60              }
61          }
62          inputEnd.connect(this);
63          System.out.println("Envoi du message à partir de " + name);
64          inputEnd.write(new Message(name, name));
65          numInstanceStart = -1;
66      }
67
68      public abstract void newInstance();
69  }
```

Listing C.15 – Utilitaire - TaskInstanceAbstract

C.2.11 TaskInstanceAtomic

La classe *TaskInstanceAtomic* est l'implémentation de la gestion des instances d'une tâche atomique.

```
1   package util;
2
3   import reo.core.InputEnd;
4   import reo.core.OutputEnd;
5
6   public class TaskInstanceAtomic extends TaskInstanceAbstract{
7
8       public TaskInstanceAtomic(OutputEnd oe, InputEnd ie, String nom) {
9           this.inputEnd = ie;
10          this.outputEnd = oe;
11          this.name = nom;
12          this.id = nom;
13          this.condition = null;
14      }
15
16      public void newInstance(){
```

```
17          instances.add(new InstanceAtomic(this, name, instances.size()));
18      }
19  }
```

Listing C.16 – Utilitaire - TaskInstanceAtomic

C.2.12 TaskInstanceComposite

La classe *TaskInstanceComposite* est l'implémentation de la gestion des instances d'une tâche composée.

```
1  package util;
2
3  import reo.core.InputEnd;
4  import reo.core.OutputEnd;
5
6  public class TaskInstanceComposite extends TaskInstanceAbstract{
7      private TaskComposite taskComposite;
8
9      public TaskInstanceComposite(TaskComposite tc, OutputEnd oe, InputEnd ie, String
           nom) {
10         this.taskComposite = tc;
11         this.inputEnd = ie;
12         this.outputEnd = oe;
13         this.name = nom;
14         this.id = nom;
15         this.condition = null;
16     }
17
18     public void newInstance(){
19         instances.add(new InstanceComposite(this, taskComposite, name, instances.
               size()));
20     }
21 }
```

Listing C.17 – Utilitaire - TaskInstanceComposite

C.2.13 TaskRelation

La classe **TaskRelation** est l'implémentation d'une condition d'un canal entre deux tâches du workflow.

```
1  package util;
2
3  public class TaskRelation {
4      private String newElementRef;
5      private String predicate;
6
7      public TaskRelation(String newElementRef){
8          this.newElementRef = newElementRef;
9      }
10
```

```
11      public void setPredicate(String predicate){this.predicate = predicate;}
12
13      public String getPredicate(){return predicate;}
14
15      public String getNewElementRef(){return newElementRef;}
16
17      public boolean isPredicate(){
18          if (predicate.equals("true()"))
19              return true;
20          else if (predicate.equals("false()"))
21              return false;
22          else return true;  // Traitement du XPath
23      }
24
25      public static boolean isPredicate(String predicate){
26          if (predicate == null) return true;
27          if (predicate.equals("true()"))
28              return true;
29          else if (predicate.equals("false()"))
30              return false;
31          else return false;  // Traitement du XPath
32      }
33  }
```

Listing C.18 – Utilitaire - TaskRelation

Table des figures

Listings